НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

«На правах рукопису»
УДК 004.421

«До захисту допущено»
Завідувач кафедри
_____ І.Р. Пархомей
(підпис)
“___”_____2019 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: Гарантований порядок доставки повідомлень в хмарних системах

Виконала: студентка другого курсу, групи ІТ-84мп
(шифр групи)
Шайдурова Катерина Анатоліївна _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник ст. викладач, к.т.н. Сирота О.П. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант НК к.т.н, доцент Пасько В.П. _____ _____
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент к.т.н. доцент кафедри АУТС, Писаренко А.В. _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

Рівень вищої освіти – другий (магістерський)

Спеціальність 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ І.Р. Пархомей
(підпис)
«___»_____2019 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Шайдуровій Катерині Анатоліївні
(прізвище, ім'я, по батькові)

1. Тема дисертації «Гарантований порядок доставки повідомлень в хмарних системах»
науковий керівник дисертації ст.викладач, к.т.н. Сирота О.П.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 28 » жовтня 2019 р. № 3770-с
2. Термін подання студентом дисертації _____ _____

3. Об'єкт дослідження – узгодженість і консенсус в розподілених системах

4. Предмет дослідження – алгоритми збереження порядку доставки та обробки повідомлень в розподілених системах

5. Перелік завдань, які потрібно розробити – аналіз предметної області та існуючих рішень; аналіз алгоритмів досягнення розподіленого консенсусу та їх модифікацій; аналіз і розробка алгоритму збереження порядку доставки та обробки повідомлень; практична реалізація алгоритму для тестування його роботоздатності.

6. Орієнтовний перелік ілюстративного матеріалу – шість плакатів

7. Орієнтовний перелік публікацій – одна публікація

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Аналіз предметної області	27 жовтня 2018 р.	
2	Постановка задачі	13 грудня 2018 р.	
3	Аналіз існуючих реалізацій	22 січня 2019 р.	
5	Аналіз алгоритмів консенсусу	26 квітня 2019 р.	
6	Розробка алгоритму роботи брокера повідомлень з гарантованим порядком доставки	9 жовтня 2019 р.	
7	Маркетинговий аналіз стартап-проекту	10 листопада 2019 р.	
8	Висновки	15 листопада 2019 р.	

Студент _____ К.А. Шайдурова
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ О.П. Сирота
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

У роботі розглянуто проблему досягнення узгодженості і консенсусу в розподілених хмарних системах для гарантування порядку доставки повідомлень, показано основні особливості існуючих рішень, використаних в брокерах повідомлень, їх переваги та недоліки.

Розроблено алгоритм, що забезпечує гарантії збереження загального порядку доставки та обробки повідомлень в умовах можливої відмови частини вузлів, що складають меншість кластеру, а також механізм переконфігурації кластеру, що втратив кворум та лідера зі збереженням всіх зафіксованих повідомлень та порядку їх фіксації. Даний алгоритм може бути використаний для більш ефективного підтримання життєздатності системи та відновлення роботи протоколу консенсусу після аварійних ситуацій у вигляді відмови кворуму кластеру.

Ключові слова: розподілений консенсус, кворум, FLP, Raft, Paxos.

Розмір пояснювальної записки – 98 аркушів, містить 6 ілюстрацій, 23 таблиці, 6 додатків.

ABSTRACT

The thesis examines the problem of achieving consistency and consensus in distributed cloud systems to guarantee message ordering, shows the main features of existing solutions used in message brokers, their advantages and disadvantages.

Developed algorithm guarantees the preservation of the total order of message delivery and processing, considering possible failures of the cluster minority. This algorithm presents the mechanism of cluster reconfiguration in case of quorum and leader failure that preserves all fixed messages and their commit order. This algorithm can be used to more effectively maintain the viability of the system and resume work of consensus protocol after quorum and leader failure.

Keywords: distributed consensus, quorum, FLP, Raft, Paxos.

Explanatory note size - 98 pages, contains 6 illustrations, 23 tables, 6 appendices.

Пояснювальна записка
до магістерської дисертації

на тему: *Гарантований порядок доставки повідомлень в хмарних системах*

Київ – 2019 року

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ

ACID – (англ. Atomicity, Consistency, Isolation, Durability) – це набір властивостей, що гарантують надійну роботу транзакцій бази даних навіть за умови наявності збоїв: атомарність, узгодженість, ізольованість, довговічність.

AMQP (англ. Advanced Message Queuing Protocol) – відкритий стандарт протоколу прикладного рівня для проміжного програмного забезпечення, орієнтованого на обробку повідомлень.

Дедуплікація (лат. deduplicatio – усунення дублів) – це процес, який направлений на виявлення та заміну цілком однакових за змістом блоків інформації одним їхнім примірником.

Дрейф годин – явище, коли через деякий час годинник «відхиляється» або поступово десинхронізується від іншого годинника.

FIFO (англ. first in, first out) – перший прийшов перший вийшов – є загальний принцип накопичення та обробки завдань (об'єктів). Принцип пов'язаний з поняттям черги: хто перший прийшов – той перший отримав обслуговування.

FLP неможливість – математично доведена Фішером, Лінч і Патерсоном відсутність рішення консенсусу у повністю асинхронній системі, яке б допускало один або більше збоїв у роботі.

RPC (англ. Remote procedure call, RPC) – протокол, що дозволяє програмі, запущеній на одному комп'ютері, звертатись до функцій (процедур) програми, що виконується на іншому комп'ютері, подібно до того, як програма звертається до власних локальних функцій.

MTBF (англ. Mean time between failures, MTBF) – відношення сумарного наробітку відновлюваного об'єкта до математичного сподівання числа його відмов протягом цього наробітку.

M2M (машино-машинна взаємодія, англ. Machine-to-Machine) – загальна назва технології, яка дозволяє забезпечити передачу даних між різними пристроями.

Офсет – зміщення всередині масиву або іншого об'єкта структури даних якуе являє собою ціле число, яке вказує відстань (зміщення) між початком об'єкта і даними елементом.

Реплікація – це механізм розподілу даних за вузлами, що дозволяє зберігати копії тих самих даних на різних вузлах мережі з метою прискорення пошуку і підвищення стійкості до відмов.

RTT (англ. Round-trip time) – в телекомунікації це час, потрібний для пересилання сигналу від передавача до отримувача, а потім у зворотньому напрямку, для підтвердження отримання сигналу

Снепшот – моментальний знімок, копія файлів і каталогів файлової системи на певний момент часу.

Split-brain (дослівно «синдром розділенного мозку») – стан розподіленої системи, коли за певних причин вона розпадається на дві чи більше частин і кожна частина продовжує функціонувати незалежно як окремий кластер, в результаті чого виникають колізії даних, що ускладнює, або навіть робить неможливим їх зворотнє об'єднання в єдину систему.

2PC – двофазний протокол фіксації – тип протоколу атомарної фіксації.

ВСТУП

Існує чимала кількість сфер діяльності, в яких при автоматизації їх бізнес-процесів за умов використання мікросервісної архітектури на перший план виходить необхідність гарантувати загальний порядок повідомлень на всіх етапах – відправки видавцем, отримання чергою повідомлень, зчитування і обробки споживачем. Прикладом може слугувати банківська сфера з її потребою чітко відслідковувати послідовність операцій з рахунком клієнта, а також будь-які інші облікові операції в бухгалтерії, в роботі аукціонів, тендерів і таке інше.

В розподілених системах випадок, коли існує потреба гарантування порядку доставки повідомлень, є найскладнішим. Будь-яке рішення, яке забезпечує жорстке впорядкування повідомлень, негативно впливає на продуктивність та пропускну здатність системи. Більше того, для забезпечення збереження порядку, в якому повідомлення надійшли до черги повідомлень, важливо також забезпечувати відмовостійкість системи за рахунок реплікації даних в кластері. Вузли кластеру за умови відмови вузла-лідера повинні продовжити роботу без перебоїв та втрат вже зафіксованих лідером повідомлень. Таким чином вибір найбільш оптимального алгоритму гарантування порядку доставки повідомлень і його модифікація для вирішення практичних проблем системи при роботі з брокером повідомлень є актуальним питанням і потребує дослідження.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ

1.1. Об’єкт та предмет дослідження

Щоб розподілена система працювала, потрібен спосіб визначення порядку подій. Однак у наборі вузлів, які працюють одночасно, неможливо сказати, яка з двох подій трапилась раніше, оскільки вузли просторово розділені. Натомість, при передачі даних через один вузол в рамках черги повідомлень або розділу теми повідомлень, постає проблема реплікації даних на декілька вузлів задля запобігання втраті повідомлень та збереження порядку доставки в разі відмови одного з вузлів.

Таким чином, однією з найважливіших абстракцій для розподілених хмарних систем є консенсус. Розподілений консенсус забезпечує узгодженість даних між вузлами розподіленої системи та досягнення згоди щодо вибору запропонованих значень. Лише за допомогою координації роботи вузлів хмарної системи із застосуванням алгоритмів консенсусу можливо надати гарантії збереження загального порядку доставки повідомлень.

Об’єктом дослідження магістерської дисертації є узгодженість і консенсус в розподілених системах.

Предмет дослідження – алгоритми збереження порядку доставки та обробки повідомлень в розподілених системах.

1.2. Аналіз роботи брокерів повідомлень

Брокер повідомлень – це посередницький програмний модуль, який переводить повідомлення з формального протоколу обміну повідомленнями відправника до формального протоколу обміну повідомленнями приймача. Брокери повідомлень – це елементи в телекомунікаційних або комп'ютерних

мережах, де програмні додатки спілкуються, обмінюючись формально визначеними повідомленнями.

У архітектурі публікації/підписки для зв'язку між машиною та машиною (M2M) в Інтернеті речей (IoT) це означає, що брокер повідомлень грає роль посередника та роз’єднує зв'язок між пристроями, які публікують інформацію, та пристроями, які на цю інформацію підписуються[27].

Така архітектура дозволяє пристрою-видавцю не знати нічого про підписантів; він повинен лише надсилати повідомлення брокеру, а потім брокер обробляє та розповсюджує їх. Повідомлення завантажуються в чергу, щоб одержувач мав змогу обробити всі вхідні повідомлення без ризику перенавантаження.

Черга повідомлень – це форма асинхронної комунікації між сервісами, що застосовується в безсерверних і мікросервісних архітектурах. Повідомлення зберігаються в черзі, поки не будуть оброблені і видалені. Кожне повідомлення обробляється тільки один раз і тільки одним споживачем. Черги повідомлень можуть використовуватися для роз'єднання складних процесів обробки, для буферизації або організації пакетної обробки, а також для згладжування пікових навантажень.

У сучасній хмарної архітектурі додатки поділяють на невеликі незалежні елементи, які легше розробляти, розгортати і обслуговувати. Черги повідомлень забезпечують для таких розподілених додатків можливість взаємодії і координації.

Черги повідомлень забезпечують наступні переваги:

1. Підвищену продуктивність.

Черги повідомлень забезпечують асинхронну взаємодію. Це означає, що кінцеві точки, які відправляють і отримують повідомлення, взаємодіють з чергою, а не один з одним. Джерела можуть додавати запити в чергу, не чекаючи їх обробки. Одержувачі обробляють повідомлення тільки тоді, коли вони доступні. Жоден компонент в системі не перебуває в очікуванні іншого, і це оптимізує потік даних.

2. Підвищену надійність.

Черги забезпечують збереження даних і зменшують кількість можливих помилок, які виникають при відключенні різних частин системи. За рахунок поділу різних компонентів за допомогою черг повідомлень можна підвищити відмовостійкість. Якщо одна частина системи стала недоступна, інша може продовжувати взаємодіяти з чергою. Існує можливість створити дзеркальну копію самої черги для забезпечення ще більшої доступності.

3. Точне масштабування.

Коли робоче навантаження досягає максимального значення, кілька екземплярів додатку можуть одночасно додавати запити в чергу без ризику виникнення конфлікту. При збільшенні черг через надходження запитів, можна розподілити робоче навантаження за групою одержувачів. Джерела, одержувачі та сама черга можуть масштабуватися за вимогою.

4. Спрощене роз'єднання.

Черги повідомлень допомагають усунути залежності між компонентами і значно спрощують написання коду роз'єднаних додатків. Компоненти програмного забезпечення, не обтяжені кодом взаємодії, можуть спеціалізуватися на виконанні конкретних бізнес-завдань.

Черги повідомлень – це оптимальний та простий спосіб роз'єднання розподілених систем, незалежно від того, використовуються в них монолітні додатки, мікросервіси або безсерверної архітектури.

Брокер повідомлень використовується для управління чергою навантаження або чергою повідомлень для декількох приймачів, забезпечуючи надійне зберігання, гарантовану доставку повідомлень і, в деяких реалізаціях, управління транзакціями.

Також брокер повідомлень може виконувати наступні дії:

1. Направляти повідомлення за одним або декількома напрямками.
2. Трансформовувати повідомлення.
3. Керувати каналами даних. Наприклад, кількістю реєстрацій у будь-якій системі.

4. Виконувати агрегацію повідомлень, розбивши повідомлення на декілька повідомлень та надсилаючи їх до місця призначення, а потім перекомпонувати відповіді в одне повідомлення для повернення користувачеві.
5. Взаємодіяти із зовнішнім сховищем, щоб доповнити повідомлення або зберегти його.
6. Викликати веб-сервіси для отримання даних.
7. Реагувати на події чи помилки.
8. Забезпечити маршрутизацію за вмістом та темою повідомлень, використовуючи шаблон публікації/підписки.

Оскільки підписанти та видавці ніколи не спілкуються безпосередньо один з одним, зменшується ризик атаки видавця.

Брокери повідомлень переважно використовуються за таких сценаріїв:

1. Необхідно надсилати дані багатьом сервісам, не викликаючи їх API безпосередньо в додатку.
2. Необхідно впорядковувати дії за прикладом транзакційної системи.
3. Необхідно відстежувати канали даних.

Перелічимо відомих брокерів повідомлень виділяють:

RabbitMQ – це "традиційний" брокер повідомлень, який реалізує різноманітні протоколи обміну повідомленнями. Це один з перших посередників повідомлень з відкритим кодом, який досяг значного рівня функціоналу, клієнтських бібліотек, інструментів для розробника та якісної документації. RabbitMQ спочатку був розроблений для впровадження AMQP – відкритого протоколу для обміну повідомленнями з потужними функціями маршрутизації. Хоча у Java є стандарти обміну повідомленнями, такі як JMS, проте він не підходить для додатків на інших мовах програмування, які потребують розподілених повідомлень, що суворо обмежує будь-який сценарій інтеграції, мікросервіс чи моноліт. З появою AMQP гнучкість між мовами стала реальною для брокерів з відкритим кодом.

Apache Kafka розроблений на Scala та початково використовувався у LinkedIn як спосіб підключення різних внутрішніх систем. У той час LinkedIn переходив до більш розподіленої архітектури і потребував переосмислення таких можливостей, як інтеграція даних та обробка потоків у режимі реального часу, відриваючись від попередніх монолітних підходів до цих проблем. Kafka сьогодні добре прийнята в екосистемі продуктів Apache Software Foundation та особливо корисна в архітектурі, що керована подіями.

Amazon Simple Queue Service (SQS) – це повною мірою керований сервіс черг повідомлень, за допомогою якого можна виділити і масштабувати мікросервіси, розроблені системи та безсерверні додатки. SQS пропонує два типи черг повідомлень. Стандартні черги забезпечують максимальну можливість випуску, оптимальне використання та доставку повідомлень за принципом «хоча б один раз». Другий варіант – FIFO SQS з обмеженою пропускною здатністю – гарантують, що повідомлення будуть оброблятися лише одноразово і виключно в порядку відправлення.

1.3. Проблема загального порядку передачі та розподіленого консенсусу

В галузі розподілених обчислень існує «Проблема загального порядку передачі». Загальний порядок передачі вказує, що повідомлення повинні бути доставлені всім учасникам у будь-якому порядку, доки він однаковий для всіх. Доставка повідомлень FIFO в розподіленій системі – це власне окремий випадок проблеми із додатковим обмеженням, накладеним на передачу повідомлень [21]. Дослідження проблеми загального порядку передачі допомагає зрозуміти обмеження можливих рішень проблеми доставки FIFO.

Проблема загального порядку передачі еквівалентна проблемі досягнення розподіленого консенсусу[2], тобто погодження усіма учасниками розподіленої системи єдиного порядку доставки повідомлень.

Вимоги до досягнення розподіленого консенсусу сформульовані наступним чином:

1. Значення, яке узгоджується, має бути запропоновано одним із учасників

2. Всі активні учасники (без збоїв) повинні прийняти рішення щодо вибору значення.

3. Обране значення має бути однаковим для всіх процесів.

У роботі Фішера, Лінча та Патерсона, відомої як "Проблема неможливості FLP" (походить від перших букв прізвищ авторів), автори обговорюють слабку форму консенсусу, при якій процеси стартують з певним початковим значенням і повинні досягти згоди щодо нового значення. Це нове значення має бути однаковим для всіх справних процесів.

Автори доходять висновку, що в асинхронній системі жоден протокол консенсусу не може бути повністю коректним за наявності навіть однієї помилки. Якщо ми не вводимо верхній ліміт часу процесу для завершення кроків алгоритму, і якщо помилки процесу неможливо точно визначити, дослідження FLP показує, що не існує детермінованого алгоритму для досягнення консенсусу.

Отже дослідження FLP означає, що консенсусу не завжди можна досягти в обмежений час.

Як наслідок, існуючі алгоритми розподіленого консенсусу / загального порядку передач, які гарантують успішне завершення у всіх можливих випадках, накладають деякі обмеження на розподілену систему, на якій вони працюють. Це в свою чергу обмежує кількість можливих випадків, коли вони можуть бути застосовані. Зняття обмежень означає, що алгоритм вийде з ладу в деяких випадках (що не обов'язково робить його непридатним на практиці). Щоб гарантувати FIFO / одноразову доставку, необхідно вводити

різні обмеження для розподіленої системи, наприклад синхронний транспортний шар, транзакції на стороні споживача тощо [11].

1.4. Постановка задачі

Метою роботи є розробка механізму гарантування порядку доставки повідомлення в хмарних системах, який забезпечує автоматичну переконфігурацію кластеру за умови втрати кворуму вузлів, є захищеним від split-brain та не потребує значних витрат часу на копіювання снепшотів стану вузла-лідера після відновлення роботи. Для досягнення мети необхідно вирішити наступні задачі:

1. Провести аналіз алгоритмів консенсусу в розподілених системах.
2. Провести дослідження щодо підходів застосування алгоритмів консенсусу в існуючих брокерах повідомлень і виявити слабкі місця представлених реалізацій.
3. Запропонувати власну реалізацію механізму гарантування порядку доставки повідомлень в хмарних системах.

Розроблений механізм гарантування порядку доставки повідомлень повинен відповідати наступним вимогам:

1. Системі потрібно 2N + 1 серверів, щоб витримати збій N серверів.
2. Повідомлення, надіслані одним видавником до одного розділу теми черги повідомлень, будуть додані до черги та отримані споживачем зі збереженням порядку.
3. Повідомлення, отримані одним споживачем в рамках одного розділу теми черги повідомлень будуть опрацьовані ним зі збереженням порядку отримання.
4. В разі збою N серверів необхідний механізм переконфігурації кластера зі збереженням інформації, що залишилась в пам’яті меншості вузлів.

Висновки по розділу

В першому розділі представлено аналіз роботи брокерів повідомлень як посередників між відправником та споживачем, що відповідають за маршрутизацію повідомлень та надають гарантії щодо порядку їх доставки та захисту від втрат. Також проведено аналіз проблеми загального порядку передачі та консенсусу, які існують в розподілених хмарних системах. Проаналізовано вимоги до досягнення розподіленого консенсусу. В результаті проведеного аналізу сформульована постановка задачі, наведена мета, перелічені задачі для її досягнення, а також поставлені вимоги до механізму гарантування порядку доставки повідомлень, що розробляється.

РОЗДІЛ 2. АНАЛІЗ ІСНУЮЧИХ РЕАЛІЗАЦІЙ ГАРАНТУВАННЯ ПОРЯДКУ ДОСТАВКИ ПОВІДОМЛЕНЬ В БРОКЕРАХ ПОВІДОМЛЕНЬ

2.1. Amazon Simple Queue Service: особливості механізмів гарантованого порядку доставки повідомлень та одноразової обробки

Черга повідомлень FIFO SQS гарантує, що повідомлення будуть передані в тому ж порядку, в якому вони були отримані. Як альтернатива, SQS може групувати повідомлення відповідно до ID групи та гарантувати впорядковану доставку в групі, але не між різними групами. Потім кожна група повідомлень може сама трактуватися як черга FIFO для цілей аналізу, що випливає далі). Це, однак, не гарантує доставку за стандартом FIFO для розподіленої системи за допомогою черги FIFO SQS, якщо ми визначимо систему, яка охоплює виробника / відправника -ів, а також споживача / одержувача -ів повідомлень. Наприклад, у простому випадку одного виробника та одного споживача система виглядала б так: (рис. 2.1)

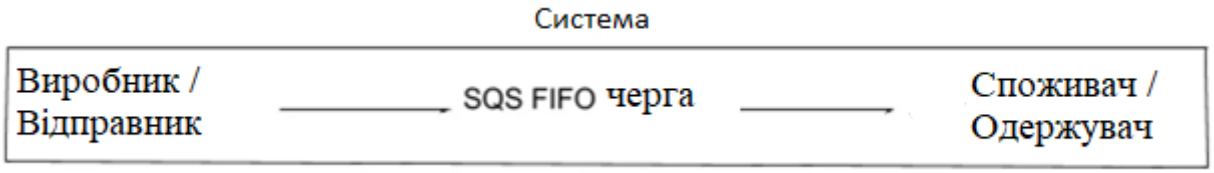


Рисунок 2.1. Робота SQS з одним виробником та споживачем

Дійсно гарантується, що повідомлення, яке потрапляє у чергу SQS FIFO, не отримуватиметься, поки у черзі є ще старі повідомлення. Це, однак, не означає, що споживач / одержувач отримає повідомлення в тому ж порядку, коли виробник / відправник подав їх до черги FIFO SQS. Щоб це було гарантовано, необхідно виконати два додаткові припущення:

1. Черга повинна отримувати повідомлення в тому ж порядку, в якому вони були вироблені виробником / відправником.

2. Споживач / одержувач повинен отримувати повідомлення в тому ж порядку, в якому вони виходять із черги SQS FIFO.

Для того, щоб ці припущення були задоволені, нам потрібен один, синхронний виробник / відправник та один, синхронний споживач / приймач, що передає дані через синхронний транспортний рівень.

Додавання асинхронності до будь-якої частини системи знімає гарантію дотримання порядку отримання повідомлень за стандартом FIFO. Як результат – гарантія впорядкованості для всієї системи також знімається. Асинхронний транспортний шар може порушити впорядкування повідомлень. Асинхронний відправник не може гарантувати порядок надсилання повідомлень. Асинхронний приймач / споживач може порушити порядок повідомлень FIFO за умови, що йому потрібно завершити обробку кожного з них у порядку FIFO. Також наявність декількох відправників або приймачів фактично можна прирівняти до наявності одного асинхронного екземпляра, тобто це також порушить загальносистемний порядок повідомлень FIFO [26].

Гарантування одноразової обробки повідомлення означає, що будь-які дубльовані повідомлення не впливатимуть на стан системи, тобто вони будуть визнані дублікатами та ігноруватимуться. Черга FIFO SQS потребує унікального ідентифікатора дедуплікації повідомлення (або він створює його, хешуючи вміст повідомлення) разом із кожним повідомленням, і використовує цей ідентифікатор для видалення повторюваних повідомлень[26]. На перший погляд здається, що використання лише черги FIFO SQS на відміну від не-FIFO буде достатньою для того, щоб вся система отримала властивість одноразової обробки. Однак існує два обмеження, запроваджені реалізацією черги SQS FIFO, які унеможливлюють гарантування одноразової обробки у всіх можливих випадках.

Перше обмеження – це максимум 5 хвилин часу на збереження заданого ідентифікатора дедуплікації повідомлення. Отже, якщо дублюване повідомлення надходить через 5 хвилин після оригіналу, SQS FIFO черга приймає його та продовжує його обробляти. Якби вимоги щодо одноразової обробки повідомлення є критичними, інші частини розподіленої системи повинні були б подбати про гарантування унікальності повідомлення.

Друге обмеження реалізації черги SQS FIFO виходить із типу гарантії "точно один раз", яку він дає споживачеві / одержувачу. Це не гарантує одноразову доставку, але гарантує одноразову обробку.

Це означає, що черга FIFO SQS повинна отримувати від споживача підтвердження, що повідомлення обробляється, перш ніж він перестане його повертати на майбутні запити повідомлень. Іншими словами, повідомлення може бути доставлено не один раз. Як це працює в реалізації черги SQS FIFO – повідомлення залишається недоступним для інших споживачів протягом певного періоду часу після його доставки до певного споживача. Це надає споживачеві, який отримав його, можливість обробити її та видалити з черги. (Видалення повідомлення з черги служить підтвердженням, що повідомлення було оброблено.) Цей проміжок часу, відомий як час очікування видимості, може тривати до 12 годин. Припущення полягає в тому, що цей період буде досить довгим, щоб дозволити отримувати та обробляти повідомлення та відправляти запит на видалення назад у чергу. Це, однак, лише припущення, і воно може бути дотримане не у всіх випадках. Отже, якщо одноразова обробка є важливою для всієї системи, факт використання черги SQS FIFO сам по собі не є гарантією. Щоб гарантувати одноразову обробку, споживачеві повідомлень доведеться вести власну незалежну “бухгалтерію”[12].

Поєднання гарантії одноразової обробки разом з FIFO має інші потенційні негативні наслідки для продуктивності для черг FIFO SQS. Щоб гарантувати замовлення FIFO, подальше повідомлення не надсилатиметься до тих пір, поки попереднє не буде визнано обробленим або не закінчиться

час його видимості. (Черга FIFO SQS доставляє групу повідомлень замість одного, але це має лише мінімальний вплив – наступна група не буде доставлена, доки черга не дізнається, що остання, яку потрібно доставити, не буде оброблена споживачем.) Таким чином стає очевидним, що пропускна здатність системи швидко погіршиться в сценаріях із низькою якістю підключення та великими тайм-аутами видимості.

Отже, що черги SQS FIFO гарантують порядок та рівномірну обробку лише в самій черзі, а не у всій розподіленій системі, яка використовувала б чергу. Черга FIFO SQS – це просто більш зручний будівельний блок для використання в розподіленій системі, спрямованій на загальну систему доставки повідомлень FIFO та одноразову обробку.

2.2. Apache Kafka: гарантії впорядкованості повідомлень

Повідомлення в Kafka поділяються на теми. Найближчою аналогією для теми є таблиця бази даних або папка у файловій системі. Теми додатково розбиваються на безліч розділів. Повідомлення пишуться до розділу лише через додавання та читаються починаючи з кінця. Тема зазвичай має декілька розділів. Впорядкування повідомлень в рамках однієї теми не гарантується – порядок додавання зберігається лише в рамках одного розділу.

Це означає, що якщо повідомлення були надіслані від виробника у визначеному порядку, брокер запише їх до розділу у такому самому порядку, і всі споживачі прочитають їх у такому самому порядку.

На високому рівні Kafka дає такі гарантії:

1. Повідомлення, надіслані виробником до певного тематичного розділу, будуть додані у порядку їх надсилання. Тобто, якщо запис M1 надсилається тим самим виробником, що і запис M2, а M1 було надіслано першим, тоді M1 буде мати менший зсув, ніж M2, і раніше з'явиться в журналі.

2. Екземпляр споживача бачить записи в порядку, в якому вони зберігаються в журналі.
3. Для теми з фактором реплікації N допускається до N-1 відмов сервера, без втрати жодних записів, внесених до журналу.

У Kafka є внутрішній процес повторного спроби: (рис. 2.2)

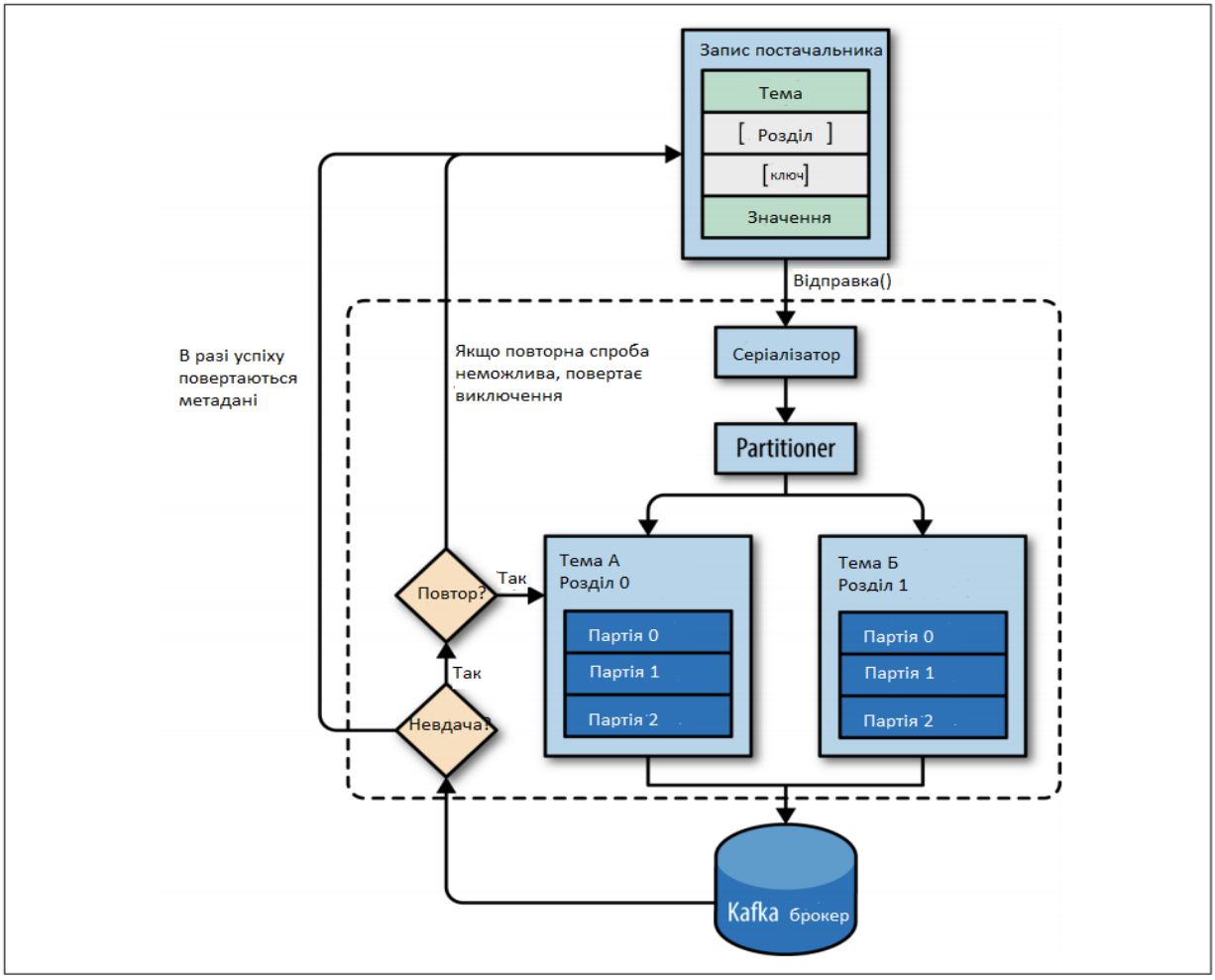


Рисунок 2.2. Механізм повторної спроби запису повідомлень брокером Apache Kafka

Якщо налаштувати кількість повторів більше нуля а themax.in.flights.requests.per.session більше, ніж один, то в разі, якщо брокер не зможе записати першу партію повідомлень, але запише другу (яка вже була в польоті), а потім повторна спроба записати першу партію буде успішною, порядок повідомлень буде порушено. Зазвичай встановлення нульової кількості повторень у надійній системі не представляється

можливим, тому якщо гарантування порядку є критичним, встановлюється in.flight.requests.per.session = 1, щоб переконатися, що, поки система робить повторну спробу надіслати пакет повідомлень, нові повідомлення не надсилатимуться (тому що це може порушити правильний порядок)[23]. Така конфігурація суттєво обмежить пропускну здатність виробника, проте гарантуватиме правильний порядок завантаження повідомлень до черги повідомлень.

Під час читання даних з розділу споживач отримує партію повідомлень, перевіряючи останній офсет в партії, а потім вимагає наступну партію, починаючи з останнього отриманого офсету. Це гарантує, що споживач Kafka завжди отримуватиме нові дані у правильному порядку, не пропускаючи жодних подій. Коли споживач зупиняється, інший споживач повинен знати, де взяти роботу – яким був останній офсет, який попередній споживач обробив, перш ніж він зупинився? "Інший" насправді може бути тим же самим споживачем, але після перезавантаження. Це не має значення – коли будь-який споживач збирається почати отримувати повідомлення з цього розділу, він повинен знати, в якому офсеті починати. Ось чому споживачі повинні зберігати свої офсети. Для кожного розділу, з якого споживач отримує повідомлення, він зберігає своє поточне місцеположення, щоб він сам або інший споживач могли дізнатися, з якого моменту продовжувати після перезавантаження. Споживачі можуть втрачати повідомлення лише за умови, якщо певний споживач зберіг офсет за зчитування повідомлень, але не встиг обробити їх до кінця. Таким чином, коли інший споживач включається до роботи, він пропустить ці повідомлення, і вони ніколи не будуть оброблені. Ось чому важливо звернути ретельну увагу на те, коли і як здійснюються офсети.

Гарантування порядку доставки зі збереженням багатороздільної структури теми досягається за допомогою спеціальних ключів в повідомленні, за якими Kafka розподіляє розміщує всі повідомлення за одним ключем на одному розділі. Розподіл здійснюється з використанням остачі від

ділення хеш-функції від ключа на кількість розділів, щоб гарантувати, що один ключ буде завжди в одному розділі. Проте зміна кількості розділів в темі порушить даний порядок.

2.3. RabbitMQ: гарантії впорядкованості повідомлень

Розділ 4.7 основної специфікації AMQP 0-9-1 пояснює умови, при яких замовлення гарантується: повідомлення, опубліковані на одному каналі, що проходять через одного агента маршрутизації, одну чергу і один вихідний канал будуть надходити в тому ж порядку, в якому вони були надіслані[24].

RabbitMQ надає більш високі гарантії з моменту випуску 2.7.0. Повідомлення можуть бути повернуті до черги за допомогою методів AMQP, які містять параметр запиту (basic.recover, basic.reject і basic.nack), або через закриття каналу, зберігаючи непідтверджені повідомлення. Будь-який із цих сценаріїв спричинив би повторне надсилання повідомлень в кінець черги для випусків RabbitMQ до 2.7.0. З випуску RabbitMQ 2.7.0 повідомлення завжди зберігаються в черзі в порядку публікації, навіть за наявності запиту чи закриття каналу.

З випуском 2.7.0 та пізніших версій, окремі споживачі все ще можуть обробляти повідомлення без збереження порядку, якщо на чергу підписано декілька споживачів. Це пов’язано з діями інших споживачів, які можуть поставити повідомлення в чергу повторно. З точки зору черги, повідомлення завжди зберігаються в порядку публікації.

Починаючи з версії 3.8 в RabbitMQ були додані черги кворуму(Quorum Queues) – черги повідомлень, які працюють з імплементацією власної варіації протоколу консенсусу Raft[8]. Черги кворуму мають на меті вирішити як проблеми продуктивності так і збої синхронізації дзеркальних черг. Raft – алгоритм розподіленого консенсусу, що означає, що це розподілений алгоритм, де кілька вузлів намагаються дійти згоди щодо певного стану. У

Raft стан – це впорядкована послідовність команд. Ці команди в RabbitMQ – це записи, читання, підтвердження тощо, які обробляє черга під час взаємодії з клієнтами.

Висновки по розділу

В другому розділі розглянуто існуючі реалізації щодо гарантування порядку доставки повідомлень в брокерах повідомлень. Описано гарантії, які надають дані брокери та визначені обмеження, які покладаються на систему доставки повідомлень у зв’язку з неможливістю безпомилкового узгодження роботи асинхронних процесів в разі відмови одного з них. Також проаналізовані механізми гарантування одноразової обробки та дедуплікації повідомлень.

РОЗДІЛ 3. АНАЛІЗ АЛГОРИТМІВ КОНСЕНСУСУ ТА ЇХ РОЛІ В ГАРАНТУВАННІ ПОРЯДКУ ДОСТАВКИ ПОВІДОМЛЕНЬ

3.1. Досягнення консенсусу в розподілених системах за допомогою алгоритму Paxos

Paxos – це консенсус-протокол, запропонований Леслі Лампорт у 1989 році. Для досягнення відмовостійкості, алгоритм Paxos пропонує використовувати набір акцепторів. Кожна пропозиція з прийняття єдиного значення буде надіслана щонайменше більшості акцепторів. Запропоноване значення буде вважатися обраним тільки коли кворум (більшість) з цих акцепторів його прийме. Навіть якщо деякі акцептори виходять з ладу, поки більшість з них функціонує, ми можемо отримати загальний результат. Якщо акцептор виходить з ладу після того, як він прийняв пропозицію, інші акцептори знають, що значення було обрано.

Якщо акцептор просто приймає перше запропоноване значення, яке він отримує, і не може змінити свій вибір, може статися ситуація, коли прийняті значення не сформують більшість. Залежно від порядку, в якому повідомлення надходять до різних акцепторів, кожен акцептор або невеликі групи акцепторів можуть прийняти різні значення, так що немає більшості акцепторів, які прийняли б однакові запропоновані значення. А отже спільне значення не може бути обрано.

З описаної ситуації випливає те, що акцептору може знадобитися змінити прийняте значення. Розглянемо варіант, коли акцептор приймає будь-яке значення, надане будь-яким заявником. Може виникнути ситуація, коли більшість акцепторів прийняла значення і воно було обране. Однак в цей же час інший сервер може запропонувати більшості акцепторів прийняти інше значення. Деякі акцептори отримають обидві ці пропозиції, оскільки прийняття значень більшістю означає, що обидва набори пропозицій мають принаймні одного спільного акцептора. Таким чином хоча б одному

акцептору доведеться змінити свою думку про те, яке значення буде обрано у кінці кінців, порушуючи цілісність. Щойно значення обрано, його зміна заборонена.

Щоб виправити це, перед тим, як пропонувати значення, заявник повинен спочатку зв’язатися з більшістю акцепторів і перевірити, чи значення вже вибрано. Якщо це так, тоді заявник повинен запропонувати вибране значення іншим акцепторам. Для імплементації цього алгоритму знадобиться двофазний протокол: спочатку перевірити, а потім надати значення.

Проте тільки перевірки недостатньо. Інший заявник може прийти після фази перевірки і запропонувати друге значення. Може виникнути ситуація, коли друге значення приймається більшістю, і після цього акцептори отримують запити від першого заявника про прийняття його значення. Протокол потрібно модифікувати, щоб гарантувати, що після того, як ми приймемо значення, конкуруючі пропозиції будуть скасовані. Для цього Paxos нумерує пропозиції. Нові пропозиції матимуть перевагу перед старими. Якщо цей перший заявник спробує закінчити свій протокол, його запити не будуть прийняті.

Paxos виділяє наступні три сутності:

1. Заявники - отримують запити (значення) від клієнтів і намагаються переконати акцепторів прийняти запропоновані ними значення.

2. Акцептори: приймають певні запропоновані значення від заявників та повідомляють заявників, якщо значення вже було прийнято раніше. Відповідь акцептора представляє собою голос за певну пропозицію.

3. Знавці: оголошують результат.

На практиці один вузол може виконувати ролі заявника, акцептора та знавця[22]. Однак для обговорення протоколу ми вважаємо, що вони є незалежними структурами.

Клієнт надсилає запит будь-якому заявнику Paxos. Потім заявник запускає двофазний протокол з акцепторами. Paxos – протокол, в якому

виграє більшості. Вимога прийняття рішень більшістю дає змогу уникнути проблем з split-brain і гарантує, що якщо значення було запропоновано і більше половини вузлів відповіли, що інших пропозицій не надходило, то це означає, що жодний інший вузол не міг запитати більше половини вузлів системи та отримати таку ж саму відповідь. Через це Paxos вимагає, щоб більшість його серверів функціонувала, щоб система могла працювати. Вимога до прийняття значення більшістю вузлів гарантує, що в разі збоїв та перезавантажень вузлів залишиться принаймні один вузол спільний від попередньої більшості до нової. Системі потрібно 2N + 1 серверів, щоб витримати збій N серверів. При цьому Paxos вимагає наявність кворуму виключно акцепторів. Кількість заявників та знавців може бути меншою.

Акцептори Paxos мають зберігати прийняті ними значення, тому їм потрібно відслідковувати інформацію, яку вони отримали від заявників, записуючи її на стабільне зберігання. Це сховища, такі як флеш-пам’ять або диск, вміст яких можна отримати, навіть якщо процес або система перезапущені.

Paxos – це двофазний протокол, що означає, що видавники взаємодіють з акцепторами двічі.

Фаза 1. Видавник запитує всіх працюючих акцепторів, чи хтось уже отримав пропозицію. Якщо відповідь ні, він пропонує значення.

Фаза 2. Якщо більшість акцепторів приймають передане значення, то консенсус досягнуто[22].

Коли заявник отримує запит клієнта для досягнення консенсусу щодо значення, він повинен створити номер пропозиції. Це число повинно мати дві властивості:

1. Бути унікальним. Жоден із двох заявників не може придумати однаковий номер пропозиції. Простий спосіб зробити це - використовувати глобальний ідентифікатор процесу для найменш значущих бітів числа. Наприклад, замість ID = 12, вузол 3 генерує ID = 12.3, а вузол 2 породжує 12.2.

2. Бути більшим, ніж будь–який раніше використаний ідентифікатор, що використовується в кластері. Заявник може використовувати монотонний лічильник. Якщо надісланий ідентифікатор не перевищує попередньо використаний, заявник дізнається про це, коли його пропозицію буде відхилено, і йому доведеться повторити спробу.

У першу фазу заявник отримує від клієнта запит на консенсус щодо значення. Він створює унікальний номер пропозиції – ID – і надсилає повідомлення `PREPARE(ID)` щонайменше більшості приймаючих.

Кожен акцептор, який отримує повідомлення `PREPARE(ID)`, дивиться на ідентифікатор у повідомленні та вирішує:

Чи більший цей ідентифікатор, ніж будь-який отриманий раніше. Якщо так – акцептор зберігає ідентифікаційний номер, і надає відповідь про успішне прийняття пропозиції `(PROMISE)`.

Якщо ні – або не надає відповіді, або повідомляє про відхилення пропозиції.

Якщо заявник отримає відповідь `PROMISE` від більшості акцепторів, то це означає, що кожен з даних акцепторів дав обіцянку, що жодна інша пропозиція з меншим ідентифікатором не може домогтися консенсусу.

В другій фазі, якщо заявник отримав повідомлення `PROMISE` від більшості акцепторів, він повинен дати команду акцепторам прийняти цю пропозицію. В противному випадку заявник ініціює початок нового раунду Paxos.

На цій фазі протоколу, заявник повідомляє всім функціонуючим акцепторам, яке значення потрібно прийняти. Він надсилає: `PROPOSE (ID, VALUE)` більшості або всім акцепторам. Кожен акцептант вирішує, чи приймати пропозицію. Він приймає пропозицію, відповідає повідомленням `ACCEPTED`, та надсилає всім знавцям повідомлення `ACCEPTED(ID, VALUE)`, якщо ідентифікаційний номер пропозиції найбільший з тих, які до нього надходили. На попередній фазі акцептор пообіцяв не приймати

пропозиції з повідомлень PREPARE з меншим ідентифікатором, але він може і прийме пропозиції з більшим ідентифікатором. Якщо знайшлась пропозиція з більшим ідентифікатором, то акцептор відповідає відмовою або не надсилає відповіді взагалі.

Повідомлення ACCEPTED може бути відправлене назад заявнику, а також знавцям, щоб вони могли поширити значення туди, де воно потрібне (наприклад, додати до журналу, змінити репліковану базу даних). Коли заявник або знавець отримує більшість прийнятих повідомлень, він знає, що консенсус щодо значення був досягнутий.

Підводячи підсумок, на першому етапі заявник з'ясовує, що жодних обіцянок щодо пропозицій із більшим ідентифікаційним номером не було. На другому етапі заявник просить акцепторів прийняти пропозицію з конкретним значенням. Поки протягом цього часу не надійшло пропозицій з більшим ідентифікатором, акцептори відповідають, що пропозиція була прийнята.

Припустимо, заявник надсилає повідомлення PREPARE з меншим ідентифікатором, ніж повідомлення, на яке більшість акцепторів відправило свої PROMISE. Деякі акцептори, ті, хто не отримав більш ранню версію ідентифікатора, можуть відповісти повідомленням з PROMISE. Але більшості голосів за це повідомлення отримано не буде, і заявник або отримає повідомлення про помилки (відмови) або вичерпає час очікування і йому доведеться повторити спробу з більш високим ідентифікатором.

Якщо акцептор ще не прийняв жодної пропозиції (тобто він відповів PROMISE на попередню пропозицію, але ще не прийняв її) він надсилає заявнику PROMISE у відповідь. Однак якщо акцептор уже прийняв попередню пропозицію, він надсилає у відповідь PROMISE з попередньо прийнятим ідентифікатором та відповідним значенням.

Акцептор повинен вести облік пропозицій, які він прийняв (на фазі 2). Якщо він вже надіслав повідомлення ACCEPT, він не може змінити свою

думку, але має повідомити заявника, що він вже прийняв пропозицію, що надійшла раніше.

Коли заявник отримує відповіді від акцепторів наприкінці фази 1, йому необхідно отримати більшість відповідей на пропозицію зі своїм ідентифікатором, щоб продовжити протокол. Він також повинен переглянути кожну із цих відповідей, щоб побачити, чи містять вони значення інших прийнятих пропозицій. Якщо таких не було, то заявник може вільно запропонувати власне значення. Якщо вони були, тоді заявник зобов'язаний вибрати значення, що відповідає найвищому ідентифікатору прийнятої пропозиції. У такий спосіб інші акцептори дізнаються про прийняті пропозиції, які вони, можливо, пропустили.

Кожен акцептор отримує повідомлення `PROPOSE(ID, VALUE)`. Якщо ідентифікатор є найбільшим з отриманих, то акцептор приймає пропозицію та повідомляє про це заявника та всіх знавців.

Якщо більшість акцепторів приймає ідентифікатор то значення, то консенсус досягнуто.

Розглянемо реакцію на можливу відмову серверів:

1. Акцептор виходить з ладу у фазі 1.

Припустимо, акцептор виходить з ладу під час фази 1. Це означає, що він не поверне повідомлення `PROMISE`. Поки заявник все ще отримує відповіді від більшості акцепторів, протокол може продовжувати діяти.

2. Акцептор виходить з ладу у фазі 2.

Припустимо, акцептор виходить з ладу під час фази 2. Це означає, що він не зможе відправити назад прийняте повідомлення. Це також не є проблемою, доки більшість акцепторів все ще функціонує і реагуватиме так, що заявник або знавець отримає відповіді від більшості акцепторів.

3. Заявник виходить з ладу на етапі `PREPARE`.

Якщо заявник виходить з ладу перш ніж надсилає будь-які повідомлення, то це те саме, як нібито він взагалі не працює.

Припустимо ситуацію, коли заявник виходить з ладу після надсилання однієї або декількох повідомлень PREPARE. Акцептор надсилає відповіді PROMISE назад, але не отримує повідомлення ACCEPT і консенсусу не буде досягнуто. Якийсь інший вузол врешті-решт запуститься як заявник, вибираючи власний ідентифікатор. Якщо вищий номер ідентифікатора працює, алгоритм працює. В іншому випадку заявник відхилить його запит і повинен вибрати більш високий ідентифікаційний номер.

4. Заявник виходить з ладу під час фази ACCEPT.

Принаймні одне повідомлення ACCEPT надіслано. Ще якийсь вузол пропонує нове повідомлення PREPARE (з більш високим ідентифікатором). Акцептор відповідає, повідомляючи цьому заявнику, що попередня пропозиція вже прийнята: PROMISE (вищий ідентифікатор, <старий ідентифікатор, значення>)

Якщо заявник отримує будь-які відповіді PROMISE зі значенням, він повинен вибрати відповідь з найвищим прийнятим ідентифікатором та змінити власне значення. Він надсилає: ACCEPT (більший ідентифікатор, значення)

Якщо заявник виходить з ладу на етапі ACCEPT, будь-який заявник, який перехоплює ініціативу, закінчує роботу з поширення старого значення, яке було прийнято.

Припустимо, більшість акцепторів отримують повідомлення PREPARE для деякого ідентифікатора, а потім повідомлення PROPOSE. Це означає, що більшість акцепторів прийняли цей ідентифікатор. Жодні PREPARE повідомлення з меншими ідентифікаторами тепер не можуть бути прийняті більшістю голосів. Для цього знадобиться більшість обіцянок для ідентифікатора з меншим номером, але акцептори вже давали обіцянки щодо ідентифікатора з більшим номером. Більшість також не приймає повідомлення PREPARE з більш високими ідентифікаторами та різними значеннями. Принаймні один акцептор знатиме ідентифікатор та відповідне

значення, яке він прийняв, і яке він поверне і поширюватиме через заявника. Ви можете мати пропозиції з більш високими ідентифікатором, але заявник зобов'язаний надати їм однакове значення.

Якщо заявник надсилає: PREPARE(найвищий ідентифікатор)

Принаймні один акцептор відповість раніше прийнятим ідентифікатором та значенням:

PROMISE (найвищий ідентифікатор, прийнятий ідентифікатор, значення)

Заявник повинен повернути PROPOSE (найвищий ідентифікатор, значення). Ось як заявники та знавці можуть дізнатися про те, що було прийнято, і в кінці кінців створити результат більшості.

Задля того, щоб мати можливість відслідковувати послідовність повідомлень, а також запобігти сценарію, за яким кожен із заявників продовжує надсилати пропозиції, кожного разу збільшуючи ідентифікатор пропозиції, жодна з яких ніколи не буде обрана. Наприклад, заявник p завершує першу фазу протоколу для номера пропозиції n1. Інший заявник q завершує першу фазу для номера пропозиції n2> n1. Заявник p не може здійснити другу фазу, тому що PROPOSE під номером n1 ігнорується, оскільки усі приймачі пообіцяли не приймати жодну нову пропозицію під номером менше n2. Отже, заявник p потім починає і завершує першу фазу для нової пропозиції номер n3> n2, що має наслідком ігнорування другої фази для заявника q. І так далі.

Щоб гарантувати прогрес, а також враховувати порядок обробки повідомлень, слід обрати репрезентативного заявника, який лише один може подавати пропозиції. Якщо репрезентативний заявник може успішно спілкуватися з більшістю акцепторів, і якщо він використовує пропозицію з номером, більшим за будь-який уже використаний, тоді його пропозиція буде прийнята. Відмовившись від пропозиції та повторивши спробу знову, якщо

він дізнається про якийсь запит із більшим ідентифікатором пропозиції, заявник врешті решт, обере пропозицію з найбільшим ідентифікатором.

Якщо достатня кількість вузлів системи (видавників, акцепторів та комунікаційна мережа) працює належним чином, життєздатність може бути досягнута вибором єдиного репрезентативного видавника. Відомий результат Фішера, Лінча та Паттерсона [20] передбачає, що надійний алгоритм вибору представника повинен здійснюватися або випадково, або за допомогою часу - наприклад, за допомогою тайм-аутів. Однак, безпека забезпечується незалежно від успіху чи невдачі виборів.

3.2. Проблеми імплементації алгоритму консенсусу Paxos

Paxos має два суттєвих недоліки. Перший недолік полягає в тому, що Paxos надзвичайно важко зрозуміти. Повне пояснення [14] є непрозорим. Як результат, було кілька спроб пояснити Paxos простішими словами [16, 18].

Друга проблема Paxos полягає в тому, що він не дає хорошої основи для побудови практичних реалізацій. Однією з причин є те, що не існує широко узгодженого алгоритму для мульти-Paxos. Описи Лампорта стосуються переважно одиночного Paxos; в його роботі окреслено можливі підходи до мульти-Paxos, але багато деталей не вистачає. Було кілька спроб розрізати та оптимізувати Paxos, таких як [9] та [25], але вони відрізняються одна від одної та від пропозицій Лампорта. Така система, як Chubby [19], реалізувала подібні до Paxos алгоритми, але в більшості випадків їх деталі не публікувалися.

Крім того, архітектура Paxos в тому вигляді, в якому він представлений в статті Лампорта, не підходить для побудови практичних систем. Наприклад, вибір окремих колекцій записів та переведення їх в послідовний журнал представляє невелику користь, а це лише додає складності. Простіше та ефективніше проектувати систему навколо журналу, де нові записи розміщуються послідовно у обмеженому порядку. Ще одна проблема полягає

в тому, що ядро Paxos – симетрична рівноправна система (хоча врешті-решт вона пропонує слабку форму лідерства для оптимізації використання)[13]. Такий підхід є корисний лише в спрощених ситуаціях, де буде прийнято лише одне рішення, але мало практичних систем використовують цей підхід. Якщо потрібно прийняти низку рішень, то простіше і швидше спочатку обрати лідера, а потім змусити лідера скоординувати рішення.

Як результат, практичні системи мало чим схожі на Paxos. Кожна імплементація починається з Paxos, розкриває труднощі в її реалізації, а потім розробляє значно іншу архітектуру. Це трудомісткий процес, схильний до помилок, а труднощі розуміння Paxos посилюють проблему. Формулювання Paxos може бути корисним для доведення теорем про його правильність, але реальні реалізації настільки відрізняються від Paxos, що докази мають мало значення. Реалізатори Chubby давали наступні коментарі щодо імплементації Paxos: Існують значні прогалини між описом алгоритму Paxos та потребами системи реального світу. Остаточна система базуватиметься на неперевіреному протоколі [4].

Через ці проблеми можна дійти висновку, що Paxos не дає хорошої основи для побудови практичної системи.

3.3. Multi-Paxos як модифікація Paxos

Алгоритм Paxos описує, як дійти до консенсусу в розподіленій системі. Однак, щоб бути корисним, розподілена система повинна мати можливість реалізувати кілька сеансів Paxos, які послідовно пов'язані між собою та формують стан системи. За термінологією Google ця модифікація визначається як Multi-Paxos [4].

Задача створення системи Multi-Paxos полягала у здійсненні повністю окремих послідовних ітерацій Paxos.

Усі вузли Paxos будуть переадресовувати запита до репрезентативного заявника, який надсилає усі пропозиції. Визначний заявник буде переходити

від однієї ітерації Paxos до наступної ітерації Paxos у послідовному порядку. Хоча у репрезентативного заявника можуть бути незавершені запити клієнтів, він буде виконувати етап підготовки до ітерації Paxos з наступним ідентифікатором, який вважає порожнім. У фазі `PROPOSE`, якщо меншість акцепторів відповіла, що значення по цьому ідентифікатору не було запропоновано, репрезентативний заявник запропонує значення, отримане від акцепторів. В іншому випадку він запропонує наступне значення з клієнтського запиту зі своєї черги. Як тільки заявник успішно отримав повідомлення про прийняття значення від більшості акцепторів, він сповіщає всі вузли про визначення значення для даної ітерації.

Система Paxos може виступати як система зберігання даних, таким чином щоб репрезентативний заявник надавав би відповідь про свій стан після імплементації поточних всіх запитів клієнта. Оскільки репрезентативному заявникові не обов'язково потрібно знати стан попередніх вирішених ітерацій перед завершенням нових (наприклад, якщо інший вузол був репрезентативним заявником для цієї ітерації, а поточний репрезентативний заявник мав не повну інформацію стосовно попередніх ітерацій), заявник може деякий час не відповідати на запити клієнтів безпосередньо після здійснення цих запитів. Натомість репрезентативний заявник спочатку дізнається про обрані на пропущених ним ітераціях значення, поки здійснює обробку нових запитів. Безпечно продовжувати здійснювати нові ітерації, збираючи інформацію про старі, тому що кожна ітерація є абсолютно окремим кроком Paxos. Як тільки дірки в ітераціях заявника були заповнені, вузол імплементує всі вирішені значення до наступної дірки. Як тільки всі незаповнені ітерації були заповнені, заявник буде знати правильний стан системи для кожної ітерації і може надсилати відповіді клієнту.

Реально функціонуючі системи Paxos, такі як Google [4], використовували паралельну реалізацію Paxos для збільшення пропускної здатності. У системі такого типу Multi-Paxos репрезентативний заявник

виконує одиночну фазу підготовки *SINGLE_PREPARE*, а потім використовує один і той самий ідентифікатор пропозицій для серії фаз пропозицій. Для забезпечення здійснення цього механізму потребується глобальний номер пропозиції для кожної ітерації. Таким чином якщо репрезентативний заявник може гарантувати, що для набору ітерацій немає значень (тобто всі перелічені номери ітерацій майбутні, на які ще не було запропоновано значення), він може виконати фазу підготовки один раз, а потім виконати кілька проходів фази пропонування з одним і тим же номером пропозиції на цих ітераціях [10]. Поки інший вузол не спробує запустити фазу підготовки з більш високим числом пропозицій (якщо вона нижча, спроба підготовки буде відхилена), репрезентативний заявник може продовжувати використовувати той самий номер пропозиції для будь-якої кількості майбутніх пропозицій. Однак, якщо інший вузол успішно завершує етап попередньої підготовки з більшим числом пропозицій, всі пропозиції, які робить поточний репрезентативний заявник, будуть відхилені. Новому репрезентативному заявнику необхідно виконати етап підготовки на достатній кількості ітерації Paxos, поки він не знайде послідовний набір ітерацій, рівний величині вікна, що використовується для послідовних пропозицій, перш ніж він може виконувати кілька фаз пропозиції без попередньої фази. Тому для репрезентативного заявника необхідно використовувати відомий розмір вікна для фаз пропозиції; в іншому випадку жоден інший вузол не міг би спокійно перервати етап підготовки.

Paxos як система забезпечує відмовостійкість та безпеку. Однак це не гарантує життєздатність системи, оскільки більше, ніж один вузол Paxos може спробувати зробити пропозицію одночасно. Пропозиція Лампорта полягала в тому, щоб система Paxos втілила ідею репрезентативного заявника для вирішення цього питання [14]. Усі вузли-заявники передають пропозиції репрезентативному заявнику, щоб мінімізувати конкуренцію в системі Paxos. Після того, як було обрано одного репрезентативного заявника, гарантується що система буде прогресувати. Однак вибір одного репрезентативного

заявника є випадком проблеми FLP. Первинно Лампортом було запропоновано використовувати фіксований порядок вибору заявника, заснований на імені вузлів [14]. Інша опція полягає у виборі репрезентативного заявника через ітерацію Paxos. Проблема першого варіанту полягає в тому, що, якщо вузол з найкращим ім'ям має дуже високу затримку або часто виходить з ладу, часто виникатиме ситуація, коли немає єдиної думки щодо того, хто репрезентативним заявником. Другий варіант виправляє цю проблему, оскільки з часом більше стійкий вузол з більшою вірогідністю стане репрезентативним заявником. Однак, поки вузли не дізнаються найбільш актуальних значень, вони не будуть знати, хто такий поточний заявник.

Таким чином, в Multi-Paxos пропонується дещо інші відмінності виборів представників, ніж безпосередньо через Paxos, щоб спростити логіку та прискорити час консенсусу. Під час звичайних операцій вузол, який робить останню успішну пропозицію, вважається репрезентативним заявником. В системі цей вузол одночасно є і репрезентативним заявником, і репрезентативним знавцем (репрезентативним знавець відповідає за вивчення вирішених значень ітерацій Paxos). Коли репрезентативний знавець вносить успішні пропозиції, він передає це рішення всім іншим вузлам. Якщо вузол розуміє, що йому не вистачає інформації про деяку ітерацію Paxos, він надсилає запит до репрезентативного знавця за інформацією. Коли кожен вузол запускається, він містить список усіх інших вузлів у системі. В цей момент він вважає, що перший вузол списку є репрезентативним знавцем.

Кожен вузол періодично надсилає повідомлення на вузол, який він вважає репрезентативним заявником, з питанням про те, кого він вважає репрезентативним заявником. Якщо у відповідь він отримує ім'я вузла, що відрізняється від імені вузла, якому надійшло питання, вузол, що робив запит, тепер вважає, що репрезентативний заявник – це цей новий вузол. Якщо репрезентативний заявник не відповідає протягом певного періоду часу, він вважає що репрезентативний заявник вийшов з ладу. Якщо декілька

вузлів Paxos намагаються стати репрезентативним заявником, є велика ймовірність, що перед тим, як один чи кілька вузлів успішно закінчать пропозицію щодо лідерства, виникне конфлікт. Отримавши повідомлення про відхилення в разі виникнення конфлікту, Paxos зробить тайм-аут для кожної відхиленої спроби, що монотонно збільшується. Якщо після закінчення тайм-ауту вузол не дізнався про те, хто є репрезентативним заявником, він збільшить ідентифікатор пропозиції, щоб бути вищим, ніж число, що спричинило його відхилення. Оскільки лише один із вузлів, що намагається стати репрезентативним заявником, не отримає повідомлення про відхилення його пропозиції (оскільки він використовував найвищий номер позиції), він повинен бути єдиним, хто намагається зробити пропозицію, а інші вузли мають чекати, поки сплине таймер тайм-ауту. Якщо цей вузол виходить з ладу або не в змозі зв’язатися з деякими вузлами Paxos, випадковий вузол повинен пропустити свою спробу стати заявником, поки інші вузли її повторюють, щоб спростити якомусь з вузлів можливість успішно завершити пропозицію та проінформувати інші вузли про те, що він є репрезентативним заявником. Важливо збільшувати тайм-аут з кожною відхиленою пропозицією, щоб переконатися, що вузол має час закінчити пропозицію (якщо затримка мережі змінюється або Paxos вузли перевантажується, час закінчення пропозиції також змінюється).

3.4. Протокол двофазної фіксації

При обробці транзакцій, в базах даних та комп'ютерних мережах двофазний протокол фіксації (2PC) – це тип протоколу атомарного запису (ACP). Це розподілений алгоритм, який координує всі процеси, що беруть участь у розподіленій атомарній транзакції, щодо того, чи слід зафіксувати чи відмінити транзакцію (це спеціалізований тип протоколу консенсусу) [3].

Цей протокол вимагає наявності координатора. Клієнт зв'язується з координатором і пропонує значення. Потім координатор намагається

встановити консенсус серед набору процесів (учасників) у дві фази, звідси і походить назва протоколу.

1. На першому етапі координатор зв'язується з усіма учасниками, пропонуючи значення, запропоноване клієнтом, і вимагає їх відповіді (фаза підготовки).

2. Отримавши всі відповіді, координатор приймає рішення здійснити запис, якщо всі учасники погодилися з запропонованим значенням або відмінити запис, якщо хтось не згоден.

3. На другому етапі координатор знову зв'язується з усіма учасниками та повідомляє про прийняте рішення (рис. 3.1).

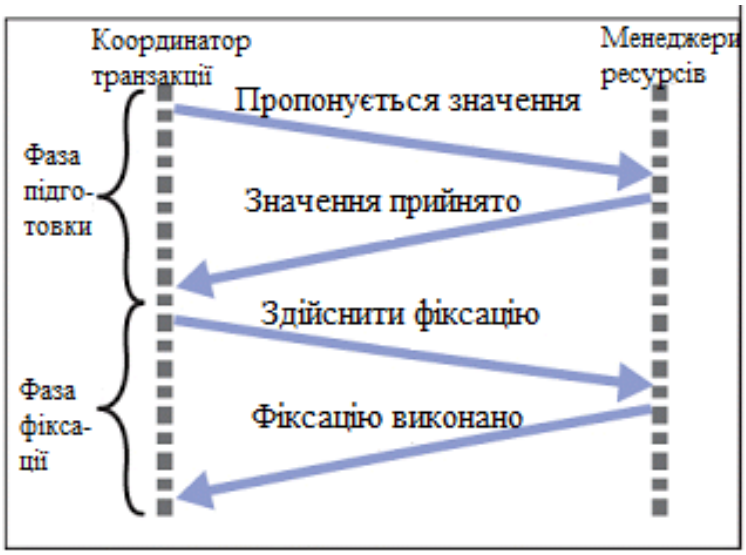


Рисунок 3.1. Схема роботи протоколу двофазної фіксації

Ми можемо бачити, що всі вищезазначені умови виконуються. Угода існує, тому що учасники лише приймають рішення "так" або "ні" щодо запропонованого координатором значенням, а самі значень не пропонують. Прийняте рішення про запис або переривання запису є чинним, тому що воно є однаковим і передається координатором до всіх учасників. Кінцевий строк виконання гарантується лише в тому випадку, якщо всі учасники передадуть відповіді координатору. Однак протокол є схильним до відмов.

Якщо говорити про збої, то які типи відмов вузла існують?

1. Режим відмови. Вузли просто виходять з ладу і взагалі не відновлюють свою роботу.
2. Режим відмови-відновлення Не вдалося відновити модель, вузли виходять з ладу та відновлення через певний час і продовжують виконувати.
3. Візантійська невдача, Вузли починають вести себе довільно, намагаючись перервати регулярну поведінку.

Тепер переходимо до того, як відмови впливають на 2-х фазову фіксацію:

1. Координатор виходить з ладу навіть перед початком фази 1. Це буквально означає, що консенсус взагалі не починається і теоретично протокол працює правильно.

2. Координатор виходить з ладу після ініціювання фази 1. Деякі вузли отримали повідомлення від координатора, що розпочинає новий раунд 2PC. Ці вузли, можливо, надіслали свої відповіді та заблоковані в очікуванні початку 2-ї фази 2PC. Це також означає, що ніякі майбутні раунди консенсусу 2PC не можуть розпочатися. Одним із способів вирішення цього питання є наявність тайм-аутів на очікування відповідей. Отже, коли тайм-аут очікування відповіді від координатора вузла збігає, він може припустити, що координатор перестав функціонувати і взяти на себе роль координатора. Він може відновити фазу 1 і зв’язатися з усіма іншими вузлами, надіславши запит на досягнення консенсусу, виходячи з значення, за яке цей вузол проголосував як учасник, до того, як фактичний координатор зазнав аварії.

Однак якщо інший вузол виходить з ладу перш ніж новий координатор збере всі повідомлення фази 1, протокол не може продовжуватися. Це пояснюється тим, що новий координатор не знає, чи встиг аварійний вузол отримати повідомлення про здійснення або скасування запису від попереднього координатора. Якщо всі інші вузли учасника погодилися здійснити запис, але аварійний вузол перестав відповідати, можливо, він

отримав директиву скасувати запис. Тож новий координатор не може прийняти рішення здійснити запис або навпаки, відмінити його.

Інша велика проблема полягає в тому, що якщо вузол учасника виходить з ладу під час фази фіксації, координатор не знає, аварійний вузол вийшов з ладу до здійснення запису чи після цього. Отже координатор не може з власної ініціативи вирішувати, чи буде здійснена транзакція.

Підсумовуючи вищеописане можна дійти висновку, що одним з найбільших недоліків двофазної фіксації є те, що це блокуючий протокол. Якщо вузли кластера надсилають координатору повідомлення про згоду, вони утримують на зберіганні ресурси, пов'язані з консенсусом, доки не отримують від координатора повідомлення про запис або відміну запису. Вихід з ладу координатора не дозволить вузлам відновити свою роботу. Те ж саму стосується відмови вузла.

3.5. Протокол трьохфазної фіксації

Основна мотивація алгоритму 3PC полягає в тому, щоб вирішити той факт, що двофазний протокол фіксації блокується у разі відмови координатора чи учасника.

Обробка 2PC блокується, коли протокол доходить до фази підготовки, а координатор виходить з ладу або недоступний через збої в мережі[6]. Немає способу закінчити виконання протоколу, поки координатор та всі учасники не відновлять роботу. Іншими словами, учасник не може досягти остаточного стану за умови відмови.

Задля вирішення цієї проблеми можна було б запустити нового координатора, відповідального за відновлення роботи протоколу і очікувати, що він дізнається, як закінчити транзакцію з оточення навколо, але це можливо лише за умови, коли всі жодний вузол не вийшов з ладу. В противному випадку протягом часу очікування відновлення його роботи

учасники ув'язнені у підготовленому стані і не можуть змінити стан самостійно.

Проблему блокування двофазного протоколу фіксації було вирішено через розробку трьохфазного протоколу фіксації.

Протокол трьохфазної фіксації – це розширення двофазного протоколу, де фаза фіксації розділена на дві фази наступним чином [6].

а. Підготовка до запису. Після одноголосного отримання відповіді «так» від всіх вузлів у першій фазі 2PC координатор просить усіх учасників підготуватися до запису. Під час цього етапу всі учасники блокують ресурси, необхідні для запису, але вони фактично не здійснюють його.

б. Якщо координатор отримує відповідь «так» від усіх учасників під час фази підготовки до виконання зобов’язань, він надсилає команду усім учасникам здійснити запис.

Наслідком додавання фази підготовки до запису стало те, що для кожного вузла існує лише один остаточний стан – відхилити запис або здійснити його після підготовки до запису. Цей новий стан дає можливість уникнути того, що команда координатора на фазі підготовка до запису може варіюватись для різних вузлів в разі збоїв в роботі координатора, як це було в 2PC.

Якщо учасник перебуває у стані підготовка до запису, ми знаємо, що кожен учасник попередньо відповів «так» координатору на запит голосування, і вони чекають на команду початку запису. Через цей механізм учасники можуть спільно вирішити загальний результат транзакції у випадку відмови координатора.

Наприклад, якщо координатор відновлення, який починає діяти після відмови попереднього координатора під час другої фази 2PC, при запиті учасників дізнається, що деякі вузли перебувають у фазі запису, він передбачає, що попередній координатор перед збоєм прийняв рішення про початок запису. А отже він може продовжувати виконувати протокол. Так само, якщо учасник каже, що він не готовий до запису, то новий координатор

може припустити, що попередній координатор зазнав невдачі ще до того, як розпочав фазу підготовки до запису. Отже, можна сміливо припускати, що жоден інший учасник ще не здійснив запис змін, а значить, можна безпечно перервати транзакцію.

Описаний механізм представлено на рис. 3.2.

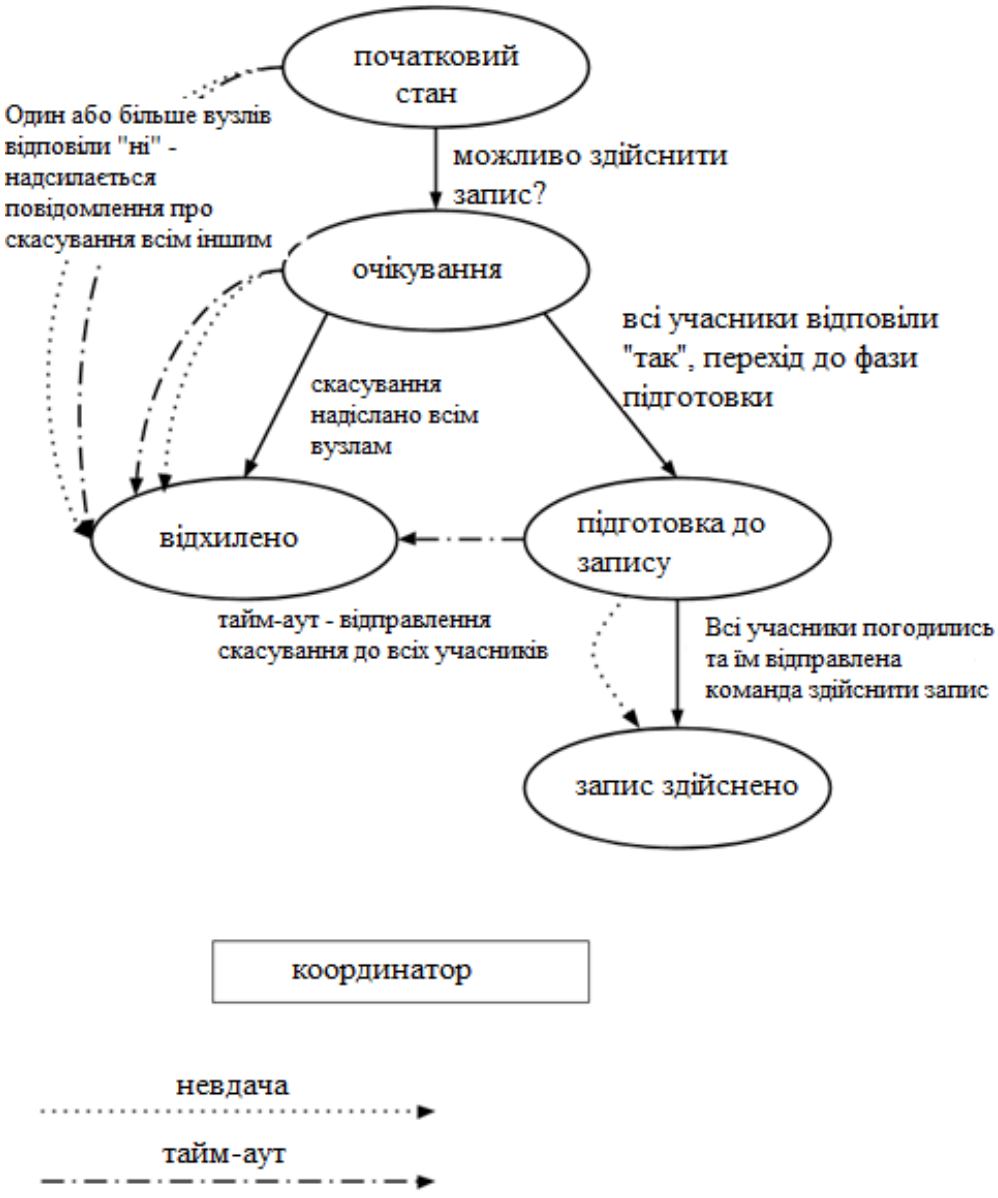


Рисунок 3.2. Схема роботи протоколу трьохфазної фіксації

Недоліком протоколу є те, що він вимагає щонайменше трьох обмінів повідомленнями в обидва кінці, а отже витрачає три RTT(Round-trip time) – час, потрібний для пересилання сигналу від передавача до отримувача, а

потім у зворотньому напрямку, для підтвердження отримання сигналу. Це потенційно затримка для завершення кожної транзакції.

Висновки по розділу

В третьому розділі розглянуті алгоритми консенсусу. Проаналізовано етапи узгодження запропонованого значення серед розподілених вузлів та реакцію алгоритму на вихід з ладу одного з вузлів кожної ролі (координатора/лідера, приймача, знавця) на кожному з етапів. Проаналізовано механізми роботи алгоритмів та визначені переваги та недоліки їх застосування.

РОЗДІЛ 4. РОЗРОБКА АЛГОРИТМУ РОБОТИ БРОКЕРА ПОВІДОМЛЕНЬ З ГАРАНТОВАНИМ ПОРЯДКОМ ДОСТАВКИ

4.1. Формулювання вимог до архітектури видавець-одержувач

Визначимо систему, яка охоплює видавця -ів, а також одержувача -ів повідомлень. В простому випадку одного видавця та одного одержувача система виглядала б так: один видавець передає повідомлення до черги повідомлень, на яку підписаний один одержувач. Визначення порядку повідомлень ускладнюється зі збільшенням кількості видавців або одержувачів. В ситуації, коли кілька одержувачів підписані на певну чергу повідомлень (або тему) навіть якщо повідомлення отримуються з теми в правильному порядку, немає гарантій, що цей порядок буде збережений, коли повідомлення будуть оброблятися одержувачами. Якщо порядок обробки важливий, то одержувачам потрібно буде координуватись через деяку систему зберігання ACID.

Аналогічно, кілька видавців, що відправляють повідомлення на одну і ту ж тему, унеможливлюють збереження порядку повідомлень.

Як визначити порядок повідомлень, що опубліковані від різних видавців? Або самі видавці повинні скоординуватися, або сама служба доставки повідомлень повинна приєднувати порядок замовлення до кожного вхідного повідомлення. Кожне повідомлення повинно містити інформацію, що визначає його місце в загальному порядку повідомлень. Це може бути або часовою позначкою (яку отримують усі сервери з одного джерела, щоб уникнути проблем розсинхронізації часу), або порядковим номером (отриманий з одного джерела з гарантіями ACID). Проте порядок отримання відповіді від вузла, який вираховує час, може не співпадати з порядком відправлення до нього запитів, а координація незалежних годинників на вузлах системи стикається з проблемою "дрейфу годин" – явища, при якому годинник відраховує час з трохи різною швидкістю [15].

Тому для вирішення проблеми гарантованої доставки повідомлень в системі видавець/споживач пропонується використовувати одного видавця, що надсилає повідомлення через один сервер (до одного підрозділу теми) до одного одержувача. Асинхронна обробка зберігається в рамках поділу теми на підрозділи, де повідомлення групуються за певним критерієм та обробляються паралельно по групах. Порядок повідомлень зберігається в рамках підрозділу.

4.2. Розробка механізму роботи брокера з гарантією порядку доставки з імплементацією алгоритму Raft

Алгоритми консенсусу використовуються в брокерах повідомлень для обрання лідера та узгодження значення реплікаційного фактору серед вузлів. Ці алгоритми гарантують збереження коректної роботи системи з 2N + 1 вузлів за умови відмови N вузлів, що безпосередньо впливає на збереження порядку доставлення повідомлень.

Для створення реплік та гарантії збереження даних при відмові вузла, де формується черга повідомлень, пропонується використати алгоритм консенсусу Raft для визначення вузла-лідера та реплікації його записів до послідовників.

Raft – алгоритм, який працює з поправкою на те, що консенсус не завжди може бути досягнуто у фіксований час. Важливою особливістю, що відрізняє Raft, є використання загального (колективного) тайм-ауту для отримання результату (прийняття рішення). У Raft, якщо стався збій і перезавантаження, необхідний період очікування як мінімум в один раунд тайм-ауту перед тим, як буде зроблена нова спроба оголосити лідера, і таким чином гарантовано успішне отримання рішення.

Результат застосування Raft еквівалентний результату застосування Paxos[5]. Raft є таким же ефективним, як і Paxos, але його структура відрізняється. Raft відокремлює ключові елементи консенсусу, такі як вибори

керівника, реплікацію журналу та забезпечення безперебійності, і забезпечує більш високий ступінь узгодженості для зменшення кількості станів системи, які необхідно враховувати. Raft також включає новий механізм зміни членства в кластері, який використовує перекриття кворумів для гарантування безпеки.

За допомогою Raft всі читання та записи проходять через лідера, завдання якого – здійснити реплікацію записів на вузли послідовників. Коли клієнт намагається зробити читання/запис з вузла послідовника, йому повідомляється, хто є лідером, і що він має надсилати всі записи до цього вузла. Лідер підтвердить клієнту те, що запис було здійснено, лише після того, як кворум вузлів-послідовників підтвердив, що вони записали дані на сервер. Кворум – це більшість вузлів. Кластер з трьох вузлів має кворум з двох, кластер з п'яти має кворум з трьох. Кожен вузол зберігає всі повідомлення в структурі журналу, де кожне повідомлення має свій індекс. Завдання алгоритму консенсусу – забезпечити, щоб усі вузли мали однакові повідомлення з однаковими індексами у своєму журналі[5].

Raft включає в себе такі основні дії:

– вибори лідера;

– реплікація журналу.

Вибори лідера – це процес узгодження лідера. Raft не допускає одночасно двох функціональних лідерів.

Кожен вузол може бути в одному з трьох станів:

Послідовник. У цьому стані вузол не видає запитів, але пасивно чекає надходження запитів від лідера чи кандидатів.

Кандидат. Використовується для виборів лідера. Коли послідовник виявить, що зв'язок з лідером втрачено, він переходить у стан кандидата та починає надсилати запити на голосування на всі інші вузли.

Лідер. Відповідальний за взаємодію з клієнтами та реплікацію журналу до послідовників.

Коли вузол запускається, він набуває статусу послідовника і чекає серцебиття від лідера. Якщо час очікування вийшов, він набуває статус кандидата і надсилає запит на голосування всім членам кластеру. Якщо інші вузли також вичерпали час очікування серцебиття лідера, вони також можуть одночасно надсилати запити на голосування. Якщо кандидат не отримає більшість голосів, то вузол переходить у стан «Послідовник». Тільки якщо вузол отримає більшість голосів, він стане лідером. Перший вузол, який надсилає свої запити на голосування, зазвичай стає лідером.

Як тільки вузол стає лідером, він посилає періодичне серцебиття всім послідовникам. Якщо лідер виходить з ладу або зупиняється, то коли в послідовників сплине період очікування отримання серцебиття від лідера, вони знову почнуть відправляти запити на голосування, і один з них стане новим лідером.

Ера – монотонний лічильник, який використовується для виявлення застарілих вузлів, які, можливо, були вимкнені деякий час і мають застарілу інформацію. Кожен раз, коли послідовник стає кандидатом, він збільшує значення своєї ери і включає її у свої запити на голосування. Після обрання лідера ера не змінюється до наступних виборів. Поточна ера включена у всі повідомлення, якими обмінюються вузли, як механізм безпеки.

Для перемоги на виборах існує три правила:

1. Більшість вузлів повинні відповісти позитивним голосом.

2. Коли кандидат надсилає запит на голосування, він включає до нього значення терміну, який він вважає поточним, + 1. Приймальний вузол перевіряє, що термін більший за його власний. Це запобігає здобуттю лідерських позицій ушкодженими вузлами (які, можливо, були відключені протягом деякого часу).

3. Кандидат включає в запит на голосування останній індекс свого журналу. Вузол буде відхиляти повідомлення, якщо індекс менший за його власний індекс останнього запису до журналу. Це гарантує, що вузол, який

не має останніх записів, не може стати лідером, оскільки це може призвести до втрати даних.

Після обрання лідера він починає обслуговувати запити клієнтів. Кожен клієнтський запит містить команду, яку необхідно виконати всіма репліками. Лідер додає команду до свого журналу в якості нового запису, а потім передає RPC `AppendEntries` паралельно кожному з інших серверів для копіювання запису. Коли запис було надійно скопійовано на інші сервери (як описано нижче), лідер застосовує запис до свого стану машини(в даному випадку – додає до черги повідомлень) і повертає клієнту результат виконання цього запиту. Якщо послідовники виходять з ладу або запускаються повільно або втрачаються мережеві пакети, лідер повторно надсилає RPC `AppendEntries` (навіть після того, як він відповів клієнтові), поки всі послідовники врешті-решт не зберігають усі записи журналу. Кожен запис журналу зберігає команду для зміни стану машини разом із номером епохи, коли запис був отриманий лідером. Номери епох в записах журналу використовуються для виявлення невідповідностей між журналами та для забезпечення деяких властивостей, серед яких: тільки оди лідер може бути присутнім за однієї епохи; якщо два журнали включають запис з однаковими індексом та епохою, це означає, що журнали ідентичні за записами до цього запису включно; якщо запис журналу зафіксовано за певною епохою, цей запис буде присутній у всіх журналах лідерів з вищою епохою.

Кожен запис журналу також має цілочисельний індекс, що визначає його положення в журналі. Лідер вирішує, коли безпечно застосувати запис журналу до стану машини (в даному випадку – додати до черги повідомлень); такий запис називається зафіксованим. Raft надає гарантії того, що зафіксовані записи довговічні, і з часом будуть виконані на всіх наявних машинах. Запис у журнал фіксується після того, як лідер, який створив запис, передав його на більшість серверів. Це також здійснює фіксацію всіх попередніх записів в журналі лідера, включаючи записи, створені

попередніми лідерами. Лідер веде облік найбільшого індексу, який, як він знає, було зафіксовано, і він включає цей індекс у RPC `AppendEntries` (включаючи серцебиття), щоб інші сервери дізнались про нього. Після того, як послідовник дізнається, що фіксацію запису журналу здійснено, він застосовує запис до своєї локальної машини (у порядку записів журналу).

Якщо два записи в різних журналах мають однаковий індекс та епоху, вони зберігають одну і ту ж команду. Якщо два записи в різних журналах мають однаковий індекс та епоху, тоді журнали ідентичні у всіх своїх попередніх записах.

Перша властивість випливає з того, що лідер створює щонайменше один запис із заданим індексом журналу в заданій епосі, а записи журналу ніколи не змінюють свого положення в журналі.

Друга властивість гарантується простою перевіркою узгодженості, що виконується за допомогою `AppendEntries`. Під час надсилання RPC, лідер включає в свій журнал індекс та епоху запису, що безпосередньо передує новим записам. Якщо послідовник не знайде запис журналу з тим самим індексом і епохою, він відхиляє нові записи. Перевірка узгодженості діє як індукційний крок: початковий порожній стан журналів задовольняє властивість збігу журналів – `AppendEntries`, а перевірка узгодженості зберігає властивість відповідності журналу щоразу, коли журнал доповнюється. Як результат, кожен раз, коли `AppendEntries` успішно повертається, лідер знає, що журнал послідовника ідентичний до його власного журналу за допомогою успішного додання нових записів.

Під час нормальної роботи журнали лідера та послідовників залишаються відповідними один одним, тому перевірка відповідності `AppendEntries` ніколи не закінчиться невдачею. Однак збої лідера можуть залишити журнали невідповідними (попередній лідер, можливо, не встиг повністю реплікувати усі записи в своєму журналі). Ці невідповідності можуть поєднуватися через низку збоїв лідерів та послідовників. У

підписника можуть бути відсутні записи, які присутні у лідера, він може мати додаткові записи, яких немає у лідера, або і те, і інше. Пропущені та сторонні записи в журналі можуть охоплювати декілька епох. У Raft лідер вирішує таку ситуацію, змушуючи послідовників продублювати власний журнал. Це означає, що конфліктні записи в журналах послідовників будуть перезаписані записами з журналу лідера.

Щоб привести журнал послідовника у відповідність із власним, лідер повинен знайти останній запис журналу, де обидва журнали узгоджуються, видалити будь-які записи в журналі послідовника після цього пункту та надіслати послідовнику всі свої записи після цього пункту. Усі ці дії відбуваються у відповідь на перевірку узгодженості, яку здійснюють за допомогою `AppendEntries` RPC. Лідер зберігає `nextIndex` для кожного послідовника, який є індексом наступного запису журналу, який лідер буде надсилати цьому послідовнику. Коли лідер є новообраним, він ініціалізує значення `nextIndex` до наступного після останнього індексу у своєму журналі. Якщо журнал послідовника не відповідає журналу лідера, перевірка на відповідність закінчиться невдачею в наступному `AppendEntries` RPC. Після отримання відповіді про невдачу лідер декрементує `nextIndex` та повторно надсилає RPC `AppendEntries`. Врешті-решт `nextIndex` дійде до значення, де журнали лідера та послідовника співпадають. Якщо це станеться, `AppendEntries` матиме успіх, видалить всі суперечливі записи в журналі послідовника та додасть записи з журналу лідера (якщо такі є).

Після того, як `AppendEntries` виконується успішно, журнал послідовника узгоджується з лідером, і він залишатиметься таким протягом решти епохи. Якщо потрібно, протокол можна оптимізувати, щоб зменшити кількість відхилених RPC `AppendEntries`. Наприклад, відхиляючи запит `AppendEntries`, послідовник може включати епоху конфліктуючого запису та перший запис свого журналу для цієї епохи. За допомогою цієї інформації лідер може декрементувати `nextIndex`, щоб обійти всі

суперечливі записи в цій епосі; Для кожної епохи з суперечливими записами буде потрібен один RPC, а не один RPC на запис. Це означає, що лідер може здійснювати реплікацію записів журналу як до послідовника, який містить майже всі останні записи, так і до послідовника, який щойно приєднався і не має даних взагалі.

Лідер ніколи не перезаписує та не вилучає записи у своєму власному журналі. Цей механізм реплікації журналу демонструє бажані властивості консенсусу.

Одержувач записує до журналу інформацію про останні опрацьовані повідомлення для того, щоб в разі його зупинки новий екземпляр одержувача мав інформацію про індекс останньої обробленої партії повідомлень та знав, з якого моменту продовжувати зчитування з черги повідомлень.

Однією з вимог до Raft є те, що коректність роботи алгоритму не повинна залежати від часу: система не повинна давати невідповідних результатів лише тому, що якась подія відбувається швидше або повільніше, ніж очікувалося. Однак доступність (здатність системи своєчасно реагувати на клієнтів) неминуче повинна залежати від часу. Наприклад, якщо обмін повідомленнями триватиме довше, ніж типовий час між аваріями сервера, часу функціювання кандидатів не вистачить для виграшу виборів; а без стійкого лідера Raft не може працювати.

Вибори лідерів – це аспект алгоритму, де дотримання термінів є дуже важливим. Raft зможе обирати та підтримувати стабільного лідера в тому випадку, якщо система буде виконувати наступні вимоги щодо термінів:

$$\textit{Час передачі} \ll \textit{тайм-аут виборів} \ll \textit{MTBF}$$

У цій нерівності час передачі – середній час, необхідний для того, щоб сервер передавав повідомлення паралельно кожному серверу в кластері та отримав від них відповіді; тайм-аут вибору – час, необхідний для голосування та вибору лідера; MTBF – це середній наробіток між відмовами для одного сервера. Час передачі повинен бути на порядок меншим за тайм-аут виборів, щоб лідери могли надійно надсилати серцебиття та запобігати

ініціації повторних виборів; зважаючи на те, що тайм-аут виборів для кожного вузла встановлюється випадково з певного фіксованого інтервалу, ця нерівність робить поділ голосів маловірогідним. Тайм-аут виборів має бути на кілька порядків меншим, ніж MTBF, щоб система стабільно працювала.

В ситуації, коли необхідно реорганізувати мережу – змінити конфігурацію, додавши або відокремивши певні вузли – вузли потенційно можуть перейти в режим split-brain. Щоб механізм зміни конфігурації був безпечним, під час переходу не повинно бути жодного моменту, коли можливо обирати двох лідерів з однаковою епохою. Будь-який підхід, коли сервери переходять безпосередньо від старої конфігурації до нової конфігурації, небезпечний. Неможливо атомарно перемикнути всі сервери відразу, тому кластер потенційно може розколотися на дві незалежні частини з кворумами під час переходу.

З метою забезпечення безпеки зміна конфігурації повинна відбуватися в дві фази [5].

Спочатку кластер переходить до перехідної конфігурації, яка називається консенсус стику. А після того, як консенсус стику був записаний кворумом, система переходить до нової конфігурації.

Консенсус стику поєднує в собі і стару, і нову конфігурацію:

1) записи журналу реплікуються на всі сервери в обох конфігураціях;

2) будь-який сервер з будь-якої конфігурації може стати лідером;

3) досягнення згоди (для виборів та запису) вимагає окремих кворумів від вузлів зі старої та нової конфігурацій. Консенсус стику дозволяє окремим серверам переходити між конфігураціями в різний час, що не загрожує безпеці. Крім того, консенсус стику дозволяє кластеру продовжувати обслуговувати клієнтські запити протягом усієї зміни конфігурації.

Конфігурації кластерів зберігаються та передаються за допомогою спеціальних записів у журналі.

Коли лідер отримує повідомлення про зміну конфігурації, він зберігає та передає послідовникам конфігурацію консенсусу стику – C<old,new>. Сервер завжди використовує найновішу конфігурацію у своєму журналі, щоб приймати рішення, навіть якщо запис кворумом послідовників ще не здійснено. Коли консенсус стику записано кворумом, лідерами можуть стати лише сервери з C<old, new> у своїх журналах.

Тепер для лідера безпечно створити запис у журналі, що описує C<new>, і передати його послідовникам. Знову ж таки, ця конфігурація набуде чинності на кожному сервері, як тільки вона з'явиться. Коли нова конфігурація була записана кворумом послідовників за правилами C<new>, стара конфігурація вже не має значення і сервери, які за новою конфігурацією мають бути відключені, можуть бути відключені.

Проте постає проблема полягає того, що видалені сервери (ті, які не входять до C<new>) можуть порушити роботу кластеру. Ці сервери не отримуватимуть серцебиття лідера, тому після закінчення часу очікування вони почнуть надсилати запит на вибори з новою епохою, і це призведе до того, що поточний лідер перейде до стану послідовника. Зрештою нового лідера буде обрано, але видалені сервери по тайм-ауту знову почнуть розсилати запит на вибори та процес повториться, що призведе до поганої доступності брокера.

Щоб запобігти цій проблемі, сервери ігнорують запит на вибори, поки вони вважають, що лідер функціонує. Зокрема, якщо сервер отримує запит на вибори в межах мінімального часу очікування виборчих слухань після отримання серцебиття від поточного лідера, він не оновлює свою епоху і не надає свій голос.

Це не впливає на звичайні вибори, де кожен сервер чекає принаймні мінімальний час очікування виборчих слухань перед початком виборів. Однак це допомагає уникнути збоїв на видалених серверах: якщо лідер може надіслати серцебиття до кластеру, тоді він не буде переобраний за рахунок запитів на вибори з більшим значенням ери.

Реалізація алгоритму Raft в роботі брокера повідомлень RabbitMq має наступний недолік: в чергах кворуму, коли вузол, на якому розміщений один із членів кластера Raft, видаляється без попередньої зміни конфігурації, існує можливість, що більше не буде кворуму для підтвердження будь-якої операції в черзі. В результаті чергу стає неможливо видалити або знову відновити її роботу, додавши новий вузол, адже немає більшості для прийняття та розповсюдження нової конфігурації. Кластер залишається без лідера і не зможе прийняти нові запити від клієнтів, а також не зможе обрати нового лідера. Без лідера це також неможливо виправити цю ситуацію, оскільки слід операції `AddServer` та RemoveServer викликаються через лідера.

Для відновлення функціонування кластеру та запобігання втраті збережених на ньому повідомлень пропонується використовувати спеціальну операцію `InitializeCluster` та додати до обов'язкових атрибутів повідомлень окрім епохи та ідентифікатору запису ще й ідентифікатор clusterId.

`ClusterId` – унікальний ідентифікатор кластеру, запобігає можливому split-brain серед вузлів, як, наприклад, в наступному прикладі:

Розглянемо кластер з 2х серверів та наступні кроки, які є повністю законними операціями, як визначено у версії Raft у публікації Онгаро у 2014.

Послідовність кроків, що ведуть до split-brain, який залишається непоміченим:

a) Два сервери S1 і S2 утворюють кластер, причому S1 є лідером. Вони фіксують x <1 і y <2, після чого відбувається роздвоєння мережі. Оскільки жоден сервер самотужки не формує кворум, лідер не вибирається і реплікація зупиняється. На даний момент у обох серверів епоха = 1, індекс = 2.

b) Щоб повернути службу в доступний стан, адміністратор повинен змінити конфігурацію кластера за допомогою виклику S1.RemoveServer (S2). (Строго кажучи, дана операція може бути викликана тільки через лідера,

однак, оскільки це не можливо для жодного сервера стати лідером у цьому стані, має бути реалізовано автоматично або через адміністратора виконання методу примушування реконфігурації для повернення до функціонального стану.) На жаль, ще один адміністратор одночасно викликає S2.RemoveServer (S1), викликаючи split-brain.

S1 фіксує z <3 і x <4. (епоха = 2, індекс = 2)

S2 фіксує z <9. (епоха = 2, індекс = 1)

c) Коли мережа відновлюється, системні адміністратори вживають заходів, щоб знову приєднати сервери один до одного, викликаючи S1.`AddServer` (S2). Потім S1 викликає RPC `AppendEntries` щоб порівняти епоху та індекс кожного з журналів серверів та відправляє до журналу S2 запис (x <4), який, як видається, відсутній, і S2 "наздоганяє".

Нарешті, S1 здійснює запис і реплікує на S2: y <5. (епоха = 3, індекс = 1). Значення, що розходиться y (епоха = 2, індекс = 1), залишається різним на двох серверах, що формує split-brain між станами машин, поки не буде перезаписано нове значення.

Щоб запобігти вказаній ситуації, нам доведеться запровадити Універсальний Унікальний ідентифікатор бази даних – `clusterId`.

В якості `clusterId` підходить будь-який псевдоунікальний ідентифікатор.

`ClusterId` генерується при створенні нового кластеру з `InitializeCluster` і є після цього постійним і однаковим значенням для всіх серверів кластеру. Його необхідно зберегти у довговічне сховище даних зберігання і значення `clusterId` повинно бути частиною резервного копіювання. Варто зауважити, що триплет (`clusterId`, епоха, індекс) однозначно та глобально визначатиме стан бази даних чи стан кластеру. Якщо дві машини кластеру знаходяться в стані (`clusterId` = x, epoch = y, index = z), вони гарантовано є однаковими.

`InitializeCluster` може бути як початковою точкою життєвого циклу кластера Raft та стану машини, так і викликатися на сервері, який вже є частиною ініціалізованого кластеру (який має один або більше членів). Викликаючи `InitializeCluster` для другого варіанту, по суті відбувається запуск нового кластер з залишків втраченого.

Життєвий цикл закінчується, коли останній сервер видаляється з кластеру. На практиці життєвого цикл закінчується просто вимкненням серверів, які знову ніколи не будуть перезапущені.

Якщо `InitializeCluster` викликається на сервері, він генерує новий `clusterId` та встановити для інших змінних – епохи та ідентифікатор запису – значення, які були до втрати кворуму. Після ініціалізації сервер сам по собі буде більшістю кластера з 1 сервером, тому він буде продовжувати обирати себе лідером.

Повторна генерація `clusterId` допомагає запобігти виникненню split-brain. У випадку, якщо `InitializeCluster` буде викликано у кількох частинах втраченого кластеру, вони все одно поділяться на окремо визначені частини, кожна зі своєю окремою `clusterId`.

Тому при виклику `InitializeCluster` дозволяється, щоб сервер був не порожнім від даних. Якщо виклик `InitializeCluster` здійснюється на непустому сервері, дані про епоху та індекс журналу зберігаються, але їх слід інтерпретувати як вихідну точку життєвого циклу нового кластеру у відриві від кластеру, до якого раніше цей журнал вівся – тепер цей журнал живе в паралельному вимірі.

Варто зауважити, що для того, щоб інший сервер став тепер частиною кластера Raft, за допомогою виклику `AddServer`, необхідно, щоб цей сервер або мав той самий `clusterId` або він повинен перебувати в порожньому та неініціалізованому стані. Не можна додати сервер, який містить деякі непов'язані дані (або стан) – доведеться спочатку видалити такий журнал / стан з нового сервера.

Зауважимо, що `InitializeCluster` як операцію що можна викликати лише на одному сервері, який потім стає єдиним членом свого нового кластеру. Це здебільшого результат архітектури Raft: це завжди лідер (або іноді кандидат) - єдиний, хто може зв’язуватися з послідовниками. Але немає процедури для зв’язку послідовників між собою.

Проте для вирішення конкретної проблеми бажано уникати зайвих повідомлень з InstallSnapshot (які займають багато часу для великих файлів снепшотів), які буде необхідно відправити, щоб сервери, які ще не втратили зв'язок між собою, бути додані до кластеру разом, замість того, щоб ініціалізувати сервер і додавати по одному за допомогою повідомлення `AddServer`. Отже пропонується після виклику `InitializeCluster` на першому сервері щойно створений `clusterId` копіюється на будь-який інший сервер з минулого кластеру, який бажано додати до нового кластеру. Для цього пропонується реалізувати відповідний метод `setClusterId` (). Важливим є те, що даний метод можна викликати лише на вузлах, які насправді мають спільну історію журналу з сервером, `clusterId` якого копіюється на них.

Після встановлення `clusterId` на другому сервері він може потім приєднатися до першого сервера через звичайний виклик `AddServer`. Оскільки `clusterId` серверів співпадають, виклик `AddServer` буде здійснено без операції InstallSnapshot.

Пропонується автоматизувати цей процес шляхом додання сервісу, який буде періодично перевіряти стан кластеру. Висновок про те, що кворум втрачено, буде робитися на основі того, що, по-перше, кворум вузлів не надішле відповідь на запит зв'язку від перевіряючого сервісу, а епоха останніх записів в журналах меншості вузлів буде значно відрізнятися від поточної епохи вузлів. Це означатиме, що відсутність відповіді від кворуму не є випадковою, адже меншість вузлів постійно ініціює перевибори з новими ідентифікаторами повідомлень про голосування і за відсутності

кворуму не може обрати лідера. В такий ситуації сервіс обере один з вузлів, що залишилися, і викличе над ним процедуру `InitializeCluster`. Генерація clusterId в даному випадку убезпечить систему від можливості того, що наряду з автоматичним процесом реконфігурації кластеру адміністратор системи почне здійснювати ті самі дії вручну. Далі сервіс ініціюватиме зміну `clusterId` вузлів, що залишилися функціонуючими, та додасть їх через виклик `AddServer`.

4.3. Порівняння запропонованого алгоритму з ZooKeeper Atomic Broadcast

Zab (ZooKeeper Atomic Broadcast) – протокол консенсусу, один із варіацій реалізації Paxos, що забезпечує роботу ядро популярної системи Apache Kafka – ZooKeeper. Zab називають протоколом атомної трансляції, оскільки він дозволяє вузлам здійснювати той самий набір транзакцій (оновлень стану) не змінюючи порядок[7].

І Zab, і Raft проводять спеціальний етап, щоб обрати лідера. І Zab, і Raft розкладають проблему консенсусу на незалежні підпроблеми: вибори лідера, реплікація журналу, безпека та відмовостійкість. Підхід з виділенням лідера серед вузлів забезпечує більш просту основу для побудови практичних систем. Зміна лідера позначається епохою $e \in N$. Нові вибори лідера збільшать e, тому всі справно функціонуючі вузли приймають лише лідера з найбільшою епохою. Після обрання лідера, в звичайному режимі, лідер пропонує та серіалізує операції клієнта в однаковому порядку в кожній репліці.

У всіх протоколах Paxos кожне вибране значення (тобто запропоноване через запит клієнта) – це запис журналу, і кожен ідентифікатор запису z має два компоненти позначаються як епоха і лічильник. Коли лідер транслює пропозицію про запис даних, кворум послідовників голосує за пропозиції та здійснює запис після того, як лідер приймає відповідне рішення. Усі варіації

протоколу Paxos гарантують збереження порядку, а саме коли команди <z, c> доставляються вузлам, всі команди <z', c> при z' < z доставляються першими незважаючи на можливий вихід з ладу лідера[1].

Алгоритм Zab має три фази, і кожен вузол може знаходитися в одній з цих трьох фаз у будь-який момент часу. Фаза виявлення – це фаза, протягом якої відбуваються вибори лідера за відомим згідно конфігурації кворумом. Вузол може бути обраний в якості лідера лише в тому випадку, якщо він має більш високу епоху або якщо всі запропоновані епохи є однаковими, відбувається за найбільш високим ідентифікатором транзакції.

На етапі синхронізації новообраний лідер синхронізує свою початкову історію попередньої епохи з усіма послідовниками [17].

Як показано на рис. 4.1, відсутність чіткої фази синхронізації спрощує алгоритмічні стани Raft, але може призвести до більш тривалого часу відновлення на практиці.

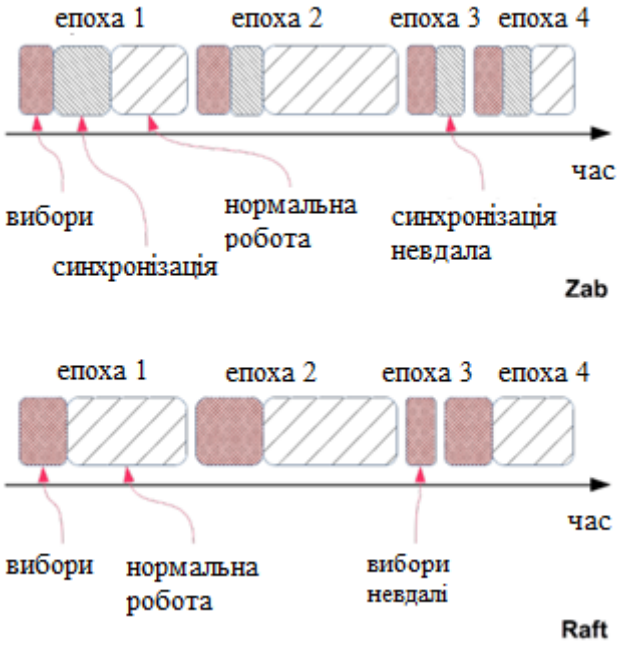


Рисунок 4.1. Механізм вибору лідера в Zab та Raft

Новообраний лідер Zab переходить до етапу трансляції лише після того, як він отримає підтвердження від кворуму, що вони синхронізувались з

ним. Фаза трансляції – це звичайний режим роботи, і лідер продовжує пропонувати дані для запису з нових запитів клієнта, поки не вийде з ладу.

На відміну від Zab, у Raft немає чіткої фази синхронізації: лідер залишається синхронізованим з кожним послідовником у фазі нормальної роботи, порівнюючи індекс журналу та значення епохи кожного запису.

Zab приймає модель обміну повідомленнями, де для кожного оновлення потрібно щонайменше три повідомлення: пропозиція, прийняття та фіксація, як показано на рис. 4.2. На відміну від Zab Raft покладається на базову систему RPC (виклику віддалених процедур). Raft також прагне мінімізувати кількість можливих станів та типи RPC, необхідних в протоколі, перевикористовуючи певні механізми. Наприклад, RPC з запитами на додання записів (`AppendEntries`) використовують лідером як для реплікації журналу, так і в якості серцебиття.

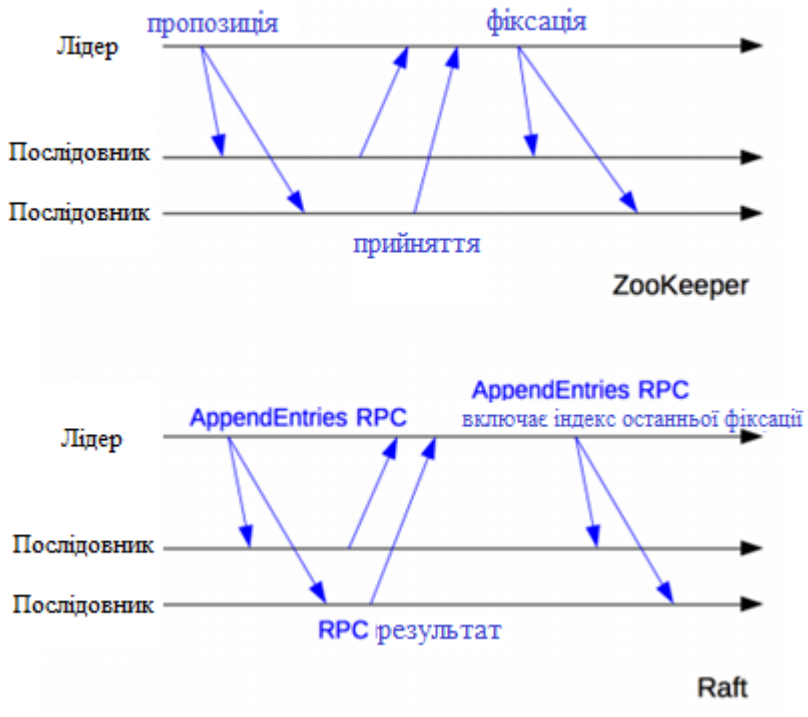


Рисунок 4.2. Механізм обміну RPC в ZooKeeper та Raft

Задача гарантування порядку доставки повідомлень в хмарних системах напряму пов'язана з розв'язанням проблеми досягнення консенсусу

та загального порядку передачі в розподілених системах. На практиці ця проблема вирішується через пом'якшення вимог до досягнення консенсусу. Існує два підходи: допускається або отримання недетермінованого консенсусу, або недосягнення консенсусу у певних випадках. Для гарантування порядку доставки для брокерів повідомлень пропонується використати схему – один видавець, що надсилає повідомлення через один сервер (до однієї теми) до одного одержувача – поєднану з алгоритмом консенсусу Raft, який використовує тайм-аути для боротьби з випадками недосягнення консенсусу.

Оригінальний Paxos був обмежений, і передбачається статичний кластер з 2f + 1 вузлів, які можуть зазнавати збоїв і відновлюватися, проте сам кластер не може розширюватися або зменшуватися. Можливість динамічно перенастроювати членство в кворумі консенсусу при збереженні послідовності даних є важливим розширенням протоколу Paxos. Динамічна конфігурація у всіх протоколах Paxos має такий базовий підхід. Клієнт пропонує спеціальну команду переналаштування з новою конфігурацією C_{new}, яка приймається як запис до журналу, як і будь-яка інша команда. Щоб забезпечити безпеку переходу до нової конфігурації, C_{new} неможливо активувати негайно та для переходу до неї необхідно пройти дві фази.

Експлуатуючи власну властивість зі збереження порядку передачі, яку пропонує Zab і Raft, обидва протоколи можуть реалізувати свої алгоритми переналаштування без обмежень для звичайних операцій або зовнішніх служб. І Zab, і Raft містять проміжну фазу, коли нові процеси в C_{new} приєднуються до кластеру в якості членів, що не мають права голосу, щоб лідер у C_{old} міг ініціалізувати їх стан шляхом передачі поточного зафіксованого префіксу оновлень. Як тільки нові процеси наздогнали лідера стартує початок переналаштування.

Різниця полягає в тому, що в Zab в час між C_{new}, запропонованим і зафіксованим, будь-які команди, отримані після переналаштування тільки

плануються до фіксації, але не будуть зафіксовані, оскільки вони мають керуватися Cnew. Однак у Raft проміжок часу вирішується кворумом $C_{old,\ new}$.

4.4. Тестування роботоспроможності алгоритму

Для перевірки роботоспроможності алгоритму розроблено додаток та здійснено тестування його роботи в умовах відключення екземплярів послідовників та лідера для імітації збоїв системи.

Очікувана поведінка – за умови втрати кворуму та за відсутності лідера, коли переконфігурація кластера неможлива, додаток має здійсните реініціалізацію кластера з вузлів, що залишились діючими, зберігши всі повідомлення клієнта, які були зафіксовані.

Статуси вузлів системи – лідер, послідовник та кандидат представлені класами додатка: `Leader`, `Follower`, `Candidate` відповідно. Контролер додатка має наступні ендпоінти: `"/message"`, `"/vote"`, `"/findLeader"`, `"/configuration"` `"/appendEntry"`, `"/configurationEntry"`. Ендпоінт `"/message"` призначений для прийняття повідомлень від клієнта якщо процес знаходиться в статусі лідера. На вхід він приймає повідомлення, та або починає процес його розповсюдження по іншим вузлам кластера, або перенаправляє до вузла лідера, якщо сам ним не є, або сповіщає про відмову у разі, якщо не має інформації про те, хто є лідером. Останній випадок вступає в дію лише після декількох кроків, а саме: якщо вузол не знає, хто є лідером, він починає процес виборів, і якщо лідера було успішно обрано, пересилає йому збережене повідомлення.

Після отримання повідомлення, лідер зберігає його у власний журнал. В додатку використано PostgreSQL для зберігання журналу повідомлень, що були отримані від користувача. Окрім тексту повідомлення в записі журналу зберігається `id` запису, епоха лідера, `clusterId` кластера, дата отримання повідомлення. Після збереження повідомлення до журналу, лідер викликає

метод sendAppendEntryToNodes(nodes, entry), який на вхід отримує адреси вузлів за конфігурацією та об'єкт Entry, який містить в собі епоху лідера, ідентифікатор лідера (щоб послідовник міг ретранслювати повідомлення, які до нього надходять, до вузла лідера), clusterId кластера, ідентифікатор та епоху попереднього запису журналу лідера (надсилається для того, щоб перевірити, чи збігаються дані журналу лідера з даними журналу послідовника), повідомлення клієнта, ідентифікатор останнього зафіксованого лідером запису (тобто запису, який був доданий до черги повідомлень). Ці дані передаються в якості RequestBody до ендпоінту вузлів "/appendEntry". Вузол, який знаходиться в статусі послідовник або кандидат надсилає у відповідь енумінатор AppendResult FAILURE або SUCCESS. Відповідь FAILURE надсилається у випадку, коли епоха, надіслана в RequestBody менша за поточну епоху вузла; коли журнал послідовника або кандидата не містить запису з попереднім індексом, який був переданий; коли запис за попереднім індексом було знайдено, проте його епоха не співпадає з надісланою попередньою епохою (в цьому випадку послідовник або кандидат видаляють конфліктуючий запис). У інших випадках послідовник або кандидат зберігають повідомлення з епохою, індексом та clusterId кластера у своїх журналах, перевіряють індекс зафіксованого запису і в разі, якщо він не співпадає з їх власним, зберігають min(індекс фіксації лідера(commitIndex), індекс останнього нового запису до журналу) та надсилають у відповідь SUCCESS.

Стан кожного вузла зберігається в базі даних та містить наступну інформацію: поточна епоха, ідентифікатор лідера, записи журналу, ідентифікатор(монотонно зростаючий) останнього доданого до черги повідомлень запису журналу, що відомий вузлу, ідентифікатор останнього запису журналу, що був перенесений до черги повідомлень.

Стан лідера: для кожного сервера зберігається nextIndex – індекс наступного запису журналу, який необхідно надіслати. В разі виникнення

конфлікту між журналом послідовника та лідера, лідер продовжує надсилати запити `AppendEntries`, поки не знайде перший запит, що співпаде, декрементуючи для цього `nextIndex`. Первинно ініціалізується до значення індексу останнього запису журналу лідера + 1.

Послідовник також зберігає дату та час останнього RPC від лідера. У разі збоїв лідера після закінчення встановленого тайм-ауту послідовник переходить до статусу кандидату, викликаючи метод `toCandidateState()`, після виклику якого він надсилає запит до ендпоінтів `"/vote"` вузлів, зазначених і конфігурації. Запит містить об'єкт `Vote`, що складається з епохи кандидата (попередньо вирахувана як поточна епоха + 1), `id` кандидата, ідентифікатор та епоха останнього запису журналу. Також в рамках `toCandidateStste()` обнуляє таймер тайм-ауту голосування. Якщо кандидат отримує запит до `"/appendEntry"`, то він знов повертається до стану послідовника `toFollower()`.

Правила для всіх вузлів системи:

Якщо індекс `commitIndex` останнього зафіксованого (доданого лідером до черги повідомлень) запису більше, ніж індекс зафіксованого самим вузлом повідомлення `lastApplied`, вузол збільшує `lastApplied` та здійснює запис до власної черги з журналу за індексом `lastApplied`.

Якщо у будь-якому запиті RPC або відповіді на нього передана епоха більше за поточну на вузлі, поточна епоха переписується на отриману, а якщо вузол був лідером, то стає послідовником.

Ендпоінт `"/configuration"` використовується для задання нової конфігурації. Якщо відповідний метод контролера викликається на вузлі, що не є лідером, він перенаправляє запит на лідера. `RequestBody` запита зберігає ідентифікатори та відповідні ним адреси вузлів, які мають формувати кластер. Лідер після отримання нової конфігурації надсилає її всім вузлам кластера, викликавши `notifyOfNewConfig(nodes)`.

Додаток включає в себе сервіс, який здійснює автоматичну переконфігурацію кластеру в разі втрати кворуму та відсутності лідера. Даний сервіс втручається в роботу протоколу, коли на запит до ендпоінту "/findLeader" він не отримує відповіді від кворуму, а вузли з меншості відповідають, що не мають інформації про діючого лідера. Також перевіряється, щоб поточна епоха вузлів, що відгукнулись, відрізнялась від епохи останнього запиту RPC, який був отриманий даним вузлом від лідера хоча б на п'ять. Це означатиме, що збій кворуму не є випадковим та вузли декілька разів намагались обрати лідера, проте через відсутність кворуму зазнали невдачі. Після цього додаток звертається до ендпоінту "/initializeCluster". Результатом стає те, що clusterId інкрементується, а вузол набуває статус лідера. В конфігурації він визначається як єдиний вузол кластеру. Після цього на інших вузлах викликається ендпоінт "/setClusterId", який встановлює новосформований clusterId кластера. Після цього вузли, які надали відповідь про успішне перевстановлення clusterId, через "/addNode", викликаний на лідері, його можна додати до конфігурації кластеру.

Було здійснено тестування додатку з імітацією відмови процесів послідовників. Здійснена фіксація втручань для реконфігурації у випадку почергової відмови вузлів від одного до чотирьох для кластеру з 7, як показано в табл. 4.1.

Таблиця 4.1. Кількість реконфігурацій системи за умови відмов вузлів

№	Кількість нефункціонуючих процесів послідовників	Кількість втручань для переконфігурації на 200 спроб
1.	1	0
2.	2	0
3.	3	0
4.	4	200
5.	5	200
6.	6	200

Як видно з таблиці, алгоритм коректно визначає необхідність переконфігурувати кластер і ініціює її за умови відмови кількості вузлів, що дорівнює числу кворуму і більше.

Висновки по розділу

В четвертому розділі описаний розроблений алгоритм роботи брокеру повідомлень. Запропонована модифікація до алгоритму консенсусу Raft для збереження даних кластеру при відновленні його роботи після втрати кворуму та лідера. Аргументовано вибір Raft як основи запропонованого алгоритму роботи брокера повідомлень через порівняння з роботою алгоритму Zab в брокері повідомлень Apache Kafka. Описано реалізацію додатку, який демонструє роботу модифікації алгоритму.

РОЗДІЛ 5. МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП ПРОЕКТУ

5.1. Опис ідеї проекту

Таблиця 5.1. Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Розробити механізм гарантування порядку доставки повідомлень в розподілених системах для брокера повідомлень, який також дозволить зберегти повідомлення після втрати кворуму.	Облікові операції: - операції за банківськими рахунками; - облік активів підприємств. Аукціони, тендери	Збереження загального порядку на етапах відправки-отримання-обробки повідомлень, відмовостійкість, можливість відновити порядок повідомлень після втрати кворуму.

Таблиця 5.2. Опис ідеї стартап-проекту

№	Техніко-економічні характеристики ідеї	Продукція конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Apache Kafka	RabbitMq			
1.	Гарантування порядку доставки	+	+	+	Впливає на масштабованість системи, бо потребує передачі через одного видавця на один сервер і отримання одним споживачем		Гарантування здійснюється в рамках розділу теми, що дає можливість паралельної обробки по групах повідомлень
2.	Швидкий перевибір вузла-лідера	+	+/-	+	Більш тривалий час відновлення у разі одночасної відмови декількох вузлів		Відсутня фаза синхронізації, що прискорює початок роботи кластеру після перевиборів

Закінчення таблиці 5.2

3.	Можливість зберегти інформацію та відновити роботу кластеру після втрати кворуму	+	+	–	Автоматичний запуск змінить фактор реплікації	Запуск механізму переформуванн я кластеру через адміністратора або автоматично	Запуск автоматичної переконфігура-ції відбудеться тільки в разі наявності великих розривів між епохою останнього запису та поточною епохою з зовнішнього сервісу

5.2. Технологічний аудит ідеї проекту

Таблиця 5.3. Технологічна здійсненність ідеї проекту

№	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Досягнення консенсусу між вузлами для реплікації та забезпечення загального порядку за допомогою модифікації Paxos	Протокол Zab у ZooKeaper, Google Chubby	+	+/- Zap є open source проектом, Chubby – ні
2	Досягнення консенсусу між вузлами для реплікації та забезпечення загального порядку за допомогою алгоритму Raft	RabbitMq починаючи з версії 3.8	+	+
Обрана технологія реалізації ідеї проекту:2				

Висновок: існує декілька варіантів реалізації ідеї проекту. Пропонується реалізація технології №3 з доданням модифікацій.

5.3. Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 5.4. Попередня характеристика потенційного ринку

№	Показники стану ринку	Характеристика
1.	Кількість головних гравців, од	Основними гравцями є RabbitMq, Apache Kafka, Amazon SQS
2.	Загальний обсяг продаж, грн./ум.од	Неможливо точно визначити. Обраховується мільйонами. RabbitMq та Apache Kafka є брокерами повідомлень з відкритим кодом. У RabbitMq є платна підтримка Pivotal та налаштування та підтримка RabbitMq з Kubernetes. Amazon SQS є платним сервісом
3.	Динаміка ринку	Зростає
4.	Наявність обмежень для входу	Відсутні
5.	Специфічні вимоги до стандартизації та сертифікації	Відсутні
6.	Середня норма рентабельності в галузі або по ринку, %	Неможливо точно визначити

Висновок: враховуючи кількість головних гравців на ринку та той факт, що більшість додатків, що будуються на взаємодії мікросервісів, використовують для їх взаємодії брокери повідомлень, можна зробити висновок, що на даний момент, ринок для входження стартап-продукту є привабливим.

Надалі визначаємо потенційні групи клієнтів, їх характеристики, потреби, особливості поведінки та сформуємо орієнтовний перелік вимог до товару для кожної групи (табл. 5.5).

Таблиця 5.5. Характеристика потенційних клієнтів стартап-проекту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці цільових груп клієнтів	Вимоги споживачів до товару
1.	Необхідно гарантувати порядок доставки повідомлень	Розробники брокерів повідомлень	Віддають перевагу певним модифікаціям Paxos або Raft.	1. Система з 2N + 1 серверів має втримувати збій N серверів. 2. Повідомлення, які надсилаються одним видавцем до конкретного розділу теми черги повідомлень будуть отримані споживачем зі збереженням порядку. 3.Необхідний механізм переконфігурації кластера в разі збою N серверів зі збереженням інформації
2.	Необхідно підтримувати життєздатність системи в разі збоїв вузлів	Розробники брокерів повідомлень	Віддають перевагу зберіганню інформації у вигляді черги або журналу	
3.	Гарантування одноразової обробки	Розробники брокерів повідомлень	Вибір між гарантування одноразового додання до черги або одноразової обробки споживачем	

Таблиця 5.6. Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1.	Конкуренти	Наявність аналогів реалізацій механізмів гарантування порядку доставки повідомлень	Доопрацювання алгоритму, власна реалізація виходу з аварійних станів кластерів черги повідомлень.
2.	Кошти на розробку та підтримку продукту	Недостатність коштів на здійснення реалізації запропонованих модифікацій	Пошук інвесторів, заклик до роботи за ідею та на перспективу; Реалізація алгоритму лише для випадку передачі повідомлень зі збереженням порядку доставки без варіацій де цей порядок є неважливим, як пропонують фірми-конкуренти
3.	Вихід аналогу	Вихід аналогічного механізму гарантування порядку доставки повідомлень може призвести до втрати актуальності	Запропонувати модифікації до існуючого рішення, що є найбільш близьким за реалізацією або здійснити швидкий вихід на ринок

Таблиця 5.7. Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1.	Новий продукт	Вихід на ринок, Зменшення монополії, Надання нових рішень у сфері	Розробка нової функціональності
2.	Вихід аналогу	Запропонувати модифікації, що зменшують кількість станів системи або надають можливість відновлення роботи після втрати кворуму вузлів	Аналіз останніх досліджень з питань досягнення консенсусу у розподілених системах, реалізація модифікацій
3.	Зворотній зв’язок від користувачів	Можливість отримання необхідної інформації для вдосконалення продукту	Наявність відгуків з боку команди розробників щодо імплементації модифікації алгоритму для вирішення можливих проблем

Таблиця 5.8. Ступеневий аналіз конкуренції на ринку

№	Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1.	Тип конкуренції: монополістична	Диференціація продукції; велика кількість продавців; відносно низькі бар'єри входження і виходу з галузі; жорстка нецінова конкуренція	Запропонована модифікація підвищує відмовостійкість брокера повідомлень порівняно з найближчими аналогами, що не покривають розглянуті випадки
2.	Рівень конкурентної боротьби: світовий	Всі існуючі реалізації розроблялись інтернаціональними компаніями та не належать до певної країни, а належать команді розробників	Пропонування клієнту функціональності, що він потребує; забезпечення більш швидкої роботи алгоритму порівняно з існуючими імплементаціями
3.	Галузева ознака: внутрішньогалузева	Механізм, що пропонується, використовується лише у сфері розробки IT додатків \ продуктів	Забезпечення підтримки роботи з середовищем розробки; Наявність детальної документації
4.	Конкуренція за видами товарів: товарно-видова	Такий тип конкуренції передбачає конкуренцію за рахунок диференціації товарів.	Впровадження модифікацій, які вирішують проблеми, рішення яких не запропоновано у товарів-замінників; Надання підтримки

Закінчення таблиці 5.8

5.	Характер конкурентних переваг: цінова та не цінова	Не цінова – представлення функціональності, якої немає у товарах-замінниках.	Впровадження унікальних модифікацій

Таблиця 5.9. Аналіз конкуренції в галузі за М. Портером

	Прямі конкуренти в галузі	Потенційні конкуренти	Постачаль-ники	Клієнти	Товари-замінники
Складові аналізу	-Невелика кількість гравців (1 бал); - темп росту ринку - високий (1 бал); - рівень диференціації продукту - товар стандартизований за ключовими властивостями, проте відрізняється за додатковими перевагами (2 бали); - жорстка цінова конкуренція а ринку(3 бали)	–	–	- 2-3 значних гравця тримають 80% ринку(3 бали); - існують мікроніші (2 бали); - високий рівень інвестицій та затрат для входу в галузь(3 бали); -доступ до каналів розподілу повністю відкритий (1 бал); - нема обмежуючих актів з боку держави (1 бал); - крупні гравці не підуть на зниження цін (2 бали); - темп росту галузі (1 бал)	Існують, але їх доля мала (2 бали)
Висновки	7 балів - рівень внутрішньогалузевої конкуренції є середнім	Загрози немає	Загрози немає	13 балів - середній рівень загрози для входу нових гравців	2 бали - середній рівень загрози

Проаналізувавши можливості роботи на ринку з огляду на конкурентну ситуацію можна зробити висновок: існуючі продукти суттєво впливають на ситуацію на ринку в цілому, проте кожен з них має свою специфічну сферу використання та свої позитивні та негативні сторони щодо рішення певних типів задач, тому робота та вихід на даний ринок є можливою і реалізованою задачею.

Для виходу на ринок продукт повинен мати функціонал що відсутній у продуктів-аналогів, повинен задовольняти потреби користувачів, мати необхідний та достатній функціонал з конфігурування, підтримку зі сторони розробників.

Таблиця 5.10. Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування
1.	Додання ідентифікатору кластеру clusterId	Можливість реініціалізувати кластер без ризику split-brain
2.	Додання можливості переконфігурації кластеру за умови втрати кворуму та лідера	Забезпечення життєздатності системи за умови неможливості виконання протоколу консенсусу за старою конфігурацією
3.	Автоматизація переконфігурації кластеру у випадку втрати кворуму та лідера	Відсутність необхідності зупиняти всі вузли для переконфігурації. Відсутність необхідності ручного втручання адміністратора
4.	Можливість ручної переконфігурації системи	У випадку, коли необхідність переконфігурації ще не була виявлена додатком, адміністратор може вручну запустити необхідний механізм, що дасть можливість зберегти зафіксовані функціонуючими вузлами повідомлення та здійснити переконфігурацію без зупинки вузлів
5.	Немає необхідності робити великі снепшоти стану лідера	Перезапис ідентифікторів вузлів кластеру clusterId дозволяє лідеру через RPC почати додавати записи починаючи з останнього записаного індексу на вузлі, а не здійснювати повне копіювання власних даних

Таблиця 5.11. Порівняльний аналіз сильних та слабких сторін запропонованого алгоритму роботи брокеру повідомлень

№	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з запропонованим						
			-3	-2	-1	0	+1	+2	+3
1.	Додання ідентифіктору кластеру clusterId	-3	+						
2.	Додання можливості переконфігурації кластеру за умови втрати кворуму та лідера	-3	+						

Закінчення таблиці 5.11

3.	Автоматизація переконфігурації кластеру у випадку втрати кворуму та лідера	-1			+				
4.	Можливість ручної переконфігурації системи	+1					+		
5.	Немає необхідності робити великі снепшоти стану лідера	-2		+					
	Загалом	-9							

Таблиця 5.12. SWOT аналіз стартап-проекту

Сильні сторони (S): – Додання можливості переконфігурації кластеру за умови втрати кворуму та лідера – Відсутність необхідності робити великі снепшоти стану лідера – Захист від split-brain за допомогою ідентифікатора кластера – Підвищення доступності системи (availability з теореми CAP)	Слабкі сторони (W): – Якщо немає резервних вузлів - зменшення фактору реплікації. Алгоритм може використовуватися у випадку, коли необхідність доступності вища за збереження встановленого фактору реплікації
Можливості (O): – Зацікавленість користувачів в збереженні availability за умови втрати кворуму	Загрози (T): – Зростання числа конкурентів; – Модифікації існуючих алгоритмів

Таблиця 5.13. Альтернативи ринкового впровадження стартап-проекту

№	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1.	Пропонування додання модифікації до імплементованого алгоритму	Головний ресурс – люди, даний ресурс – наявний	2-3 місяці
2.	Реклама	Залучення власних коштів для реклами товару	1-2 місяці
3.	Написання статей та опис алгоритму на відомих ресурсах	Головний ресурс – час, даний ресурс - наявний	2-3 тижні
4.	Презентація алгоритму на хакатонах й інших IT заходах	Ресурс – час та гроші для участі, наявні	1-3 місяці

5.4. Розроблення ринкової стратегії проекту

Таблиця 5.14. Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1.	Розробники брокерів повідомлень.	Проекти з відкритим кодом є відкритими для пропозицій, проте вони можуть не бути зацікавленими у наданні гарантій за умов втрати кворуму	Середній	Висока	Складний
2.	Розробники програмного забезпечення в банківській сфері, сфері ведення обліку, додатках для аукціонів, тендерів і т.ін.	Якщо практичні задачі вимагають продовження роботи системи за умов втрати кворуму - готові.	Високий	Середня	Середній
Які цільові групи обрано: обрано обидві цільові групи					

Відповідно до проведеного аналізу можна зробити висновок, що обидві цільові групи є підходящими для розповсюдження даного алгоритму. Хоча розробники брокерів повідомлень можуть не бути зацікавленими в допрацюванні відновлення системи після втрати лідера і кворуму, тому що гарантії роботи протоколу консенсусу базуються на коректній роботі кворуму і механізми відновлення за умов їх збою не є першочерговою задачею системи. Проте на практиці дана проблема нерідко постає і автоматизація її вирішення може зацікавити користувачів брокерів повідомлень. Те ж саме стосується і розробників програмного забезпечення в банківській сфері, сфері ведення обліку, додатках для аукціонів, тендерів і

т.ін., які можуть потребувати механізму реініціалізації кластеру після втрати кворуму.

Відповідно до стратегії охоплення ринку збуту товару обрано стратегію диференційованого маркетингу, оскільки запропонована модифікація підходить для розробки брокеру повідомлень на основі Raft, тому не може бути запропонована на всі сегменти ринку.

Таблиця 5.15. Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Надання функціональності що відсутня у товарів-замінників, підтримка клієнтів	Проведення реклами, освітлення унікальної функціональності через інтернет ресурси та інші канали, контакт напряму з споживачами; формування лояльності і прихильності споживачів	Зниження ступеню замінності товару; Прихильність клієнтів; Відмітні властивості товару; Відмітні характеристики товару	Стратегія диференціації

Таблиця 5.16. Визначення базової стратегії конкурентної поведінки

Чи є проект «першопроходцем» на ринку	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, які?	Стратегія конкурентної поведінки
Ні, оскільки є товари-замінники, але дані товари замінники не мають деякого необхідного функціоналу	Так, ціль компанії знайти нових споживачів та, частково, забрати існуючих у конкурентів задля задоволення потреб останніх	Компанія частково копіює характеристики товару конкурента, основна ціль компанії -пропозиція модифікації існуючого алгоритму, розробка нового унікального функціоналу	Стратегія заняття конкурентної ніші

Таблиця 5.17. Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
1.	Можливість переконфігурації кластеру за умови втрати кворуму та лідера	Стратегія позиціювання за перевагами продукту	- додання автоматичної переконфігурації у випадку втрати кворуму та лідера; - відсутність необхідності робити великі снепшоти стану лідера	Підвищення доступності
2.	Можливість безпечно реініціалізувати кластер	Стратегія позиціювання за перевагами продукту	- імплементація ідентифіктору кластера для запобігання ризику split-brain	Підвищення безпеки

Відповідно до проведеного аналізу можна зробити висновок, що стартап-компанія вибирає як базову стратегію розвитку – стратегію диференціації, як базову стратегію конкурентної поведінки – стратегію заняття конкурентної ніші, як базову стратегію позиціонування – стратегію позиціювання за перевагами продукту.

5.5. Розроблення маркетингової програми стартап-проекту

Таблиця 5.18. Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1.	Необхідна можливість здійснити переконфігурацію кластеру за умови втрати кворуму та лідера	Автоматизація переконфігурації	В існуючій реалізації за умови видалення вузла без попередньої переконфігурації або його збою, що приводить до втрати кворуму, якщо лідера буде втрачено, то у RabbitMq неможливо ні видалити, ні переконфікурувати вузли без зупинення їх роботи

Закінчення таблиці 5.18

2.	Необхідна можливість безпечно реініціалізувати кластер	Захист від split-brain	Якщо реконфігурація здійснюється вручну одночасно декількома адміністраторами, є шанс створення двох паралельних кластерів, які після об'єднання можуть демонструвати неконсистентність
3	Швидка реконфігурація	Не здійснювати копію снепшотів лідерського вузла	Існуюча реалізація в RabbitMq ініціалізує кластер починаючи з лідерського вузла, а всі інші вузли, які мають бути приєднані, мають мати чисті журнали
4	Швидкі перевибори лідера	Відсутність стану синхронізації вузлів	Реалізація, представлена в Apache Kafka, після обрання лідерського вузла потребує додаткового етапу синхронізації, що збільшує час, поки черга повідомлень не приймає повідомлення

Таблиця 5.19. Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td colspan="3">Сутність та складові</td></tr>
<tr><td>1. Товар за задумом</td><td colspan="3">Алгоритм гарантування порядку доставки повідомлень в брокері повідомлень</td></tr>
<tr><td rowspan="5">2. Товар у реальному виконанні</td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх/Тл/Е/Ор</td></tr>
<tr><td>Система реконфігурується за умови втрати кворуму та лідера</td><td>М</td><td>Тх/Тл</td></tr>
<tr><td>Система уникає необхідності робити великі снепшоти стану лідера за умови втрати кворуму та лідера</td><td>Нм</td><td>Тл</td></tr>
<tr><td>Система захищена від split-brain за допомогою ідентифікатора кластера при реконфігурації</td><td>М</td><td>Тл</td></tr>
<tr><td>Система швидко відновлює доступність після втрати кворуму</td><td>Нм</td><td>Тл</td></tr>
<tr><td rowspan="2">3. Товар із підкріпленням</td><td colspan="3">До продажу: наявна повна документація</td></tr>
<tr><td colspan="3">Після продажу: додаткова підтримка спеціалістів налаштування, підтримка з боку розробника</td></tr>
<tr><td colspan="4">За рахунок чого потенційний товар буде захищено від копіювання: захист інтелектуальної власності, патент</td></tr>
</table>

Таблиця 5.20. Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
До 500$ /місяць за підтримку. 0.50$ за мільйон запитів	Деякі аналоги пропонують використовувати свої реалізації безкоштовно, а беруть кошти за підтримку та додаткові налаштування у Kubernetes	Дуже диференційовані	До 500$ /місяць за підтримку

Таблиця 5.21. Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Якщо клієнти користуються стеком сервісів Амазону, то вони в переважній більшості оберуть Amazon SQS. В разі, якщо на першому місці стоятиме пропускна здатність та можливість перечитати повідомлення - оберуть Apache Kafka. В разі, якщо архітектура має бути гнучкою, конс'юмери і черги можуть змінюватись, оберуть RabbitMq	Функції обслуговування	Однорівневий канал – алгоритм можна запропонувати імплементувати в існуючу реалізацію RabbitMq	Традиційна система збуту

Таблиця 5.22. Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Клієнти цікавляться новими підходами до проблем зберігання порядку повідомлень та консенсусу	Через мережу Інтернет	Додаткова функціональність, яку пропонує модифікація	Описати проблематику, яку вирішує запропонована модифікація алгоритму	Пропонується автоматизація відновлення системи за умов втрати кворуму

Як результат було створено ринкову (маркетингову) програму, що включає в себе визначення ключових переваг концепції потенційного товару, опис моделі товару, визначення меж встановлення ціни, формування системи збуту та концепцію маркетингових комунікацій.

Висновки по розділу

В п'ятому розділі описано стратегії та підходи з розроблення стартап-проекту, визначено наявність попиту, динаміку та рентабельність роботи ринку, як висновок було вказано що існує можливість ринкової комерціалізації проекту. Розглянувши потенційні групи клієнтів, бар’єри входження, стан конкуренції та конкурентоспроможність проекту було встановлено що проект є перспективним. Розглянуто та вибрано альтернативу впровадження стартап-проекту та доведено доцільність подальшої імплементації проекту.

ВИСНОВКИ

У ході виконання магістерської дисертації було розглянуто питання гарантування загального порядку доставки повідомлень в брокері повідомлень. Були проаналізовані алгоритми консенсусу, які використовуються для реплікації повідомлень, щоб у випадку відмови лідерського вузла система могла продовжити роботу без втрат повідомлень, що вже були зафіксовані, та порядку їх відправлення. Наведено характеристику предметного середовища та обґрунтовано причину розробки алгоритму. Було проаналізовано імплементації та модифікації алгоритмів консенсусу, які були використані в існуючих реалізаціях брокерів повідомлень.

На основі даних, отриманих в процесі аналізу, сформульовано задачу запропонувати власну реалізацію механізму гарантування порядку доставки повідомлень в хмарних системах, яка б мала функціональні переваги порівняно з існуючими. Виявлено ключові відмінності модифікованих алгоритмів консенсусу та їх переваги та недоліки. Визначено практичні задачі, які не вирішуються за поточної реалізації роботи брокерів повідомлень. Розроблена модифікація алгоритму роботи брокеру повідомлень із застосуванням Raft, яка надає можливість відновлення системи після втрати кворуму. Аргументовано вибір алгоритму консенсусу Raft після порівняння з початковим варіантом Paxos, сформульованим Лампортом.

Проведений маркетинговий аналіз стартап-проекту. Проведено опис ідеї проекту, технологічний аудит ідеї проекту, аналіз ринкових можливостей запуску стартап-проекту. Розроблено ринкову стратегію проекту та маркетингову програму стартап-проекту.

Результати роботи над магістерською дисертацією опубліковані в одній статті.

Наукова новизна одержаних результатів магістерської дисертації полягає у наступному:

вперше:

- запропоновано автоматичний механізм переконфігурації кластеру, що втратив кворум та лідера;

- кінцевим користувачам надана можливість за умови наявності резервних вузлів або погодження на зменшення фактору реплікації продовжити користуватися системою навіть після втрати кворуму та лідера;

- запропоновано механізм захисту від split-brain за умови одночасної спроби здійснити переконфігурацію кластеру двома і більше учасниками (сервісом та адміністратором або кількома адміністраторами)

удосконалено:

- механізм перенесення даних з лідерського вузла після перезапуску кластера внаслідок втрати кворуму та лідера. Запропонований варіант передбачає відсутність необхідності робити великі снепшоти журналу лідера для всіх вузлів нової конфігурації. Існуюча реалізація передбачала видалення всіх записів журналу всіх вузлів окрім обраного лідером.

здобуло подальший розвиток:

- механізм доставки повідомлень FIFO в розподіленій системі;

- механізм переконфігурації кластеру для черги повідомлень;

- автоматизація переналаштування кластеру.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas. Consensus in the Cloud: Paxos Systems Demystified // IEEE Xplore, 15 September 2016
2. C. Dwork, N. Lynch, L. Stockmeyer Consensus in the Presence of Partial Synchrony // Journal of the Association for Computing Machinery, Vol. 35, No. 2, April 1988, pp. 288-323
3. Демибрас М. - Двухфазный коммит и будущее распределённых систем - [Електроний ресурс] - Режим доступу: https://muratbuffalo.blogspot.com/2018/12/2-phase-commit-and-beyond.html
4. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective // Journal of the ACM (JACM), 2007 - pp. 398–407.
5. D. Ongaro, J. Ousterhout, "In search of an understandable consensus algorithm", *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 305-319, 2014.
6. F. B. Schneider Implementing fault-tolerant services using the state machine approach: A tutorial, vol. 22, no. 4, pp. 299-319, Dec. 1990.
7. F. Junqueira, B. Reed, M. Serafini, Zab: High-performance broadcast for primary-backup systems // *Dependable Systems & Networks (DSN). IEEE*, pp. 245-256, 2011.
8. Jack Vanlightly. RabbitMQ 3.8 Feature Focus - Quorum Queues - [Електроний ресурс] - Режим доступу: https://www.cloudamqp.com/blog/2019-03-28-rabbitmq-quorum-queues.html
9. J. KIRSCH, Y. AMIR Paxos for system builders. Tech. Rep. CNDS-2008-2, Johns Hopkins University, 2008.
10. Hao Du, David J. St. Hilaire. Multi-Paxos: An Implementation and Evaluation // Semantic Scholar, 2009
11. Henry Robinson. A Brief Tour of FLP Impossibility - [Електроний ресурс] - Режим доступу: https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/

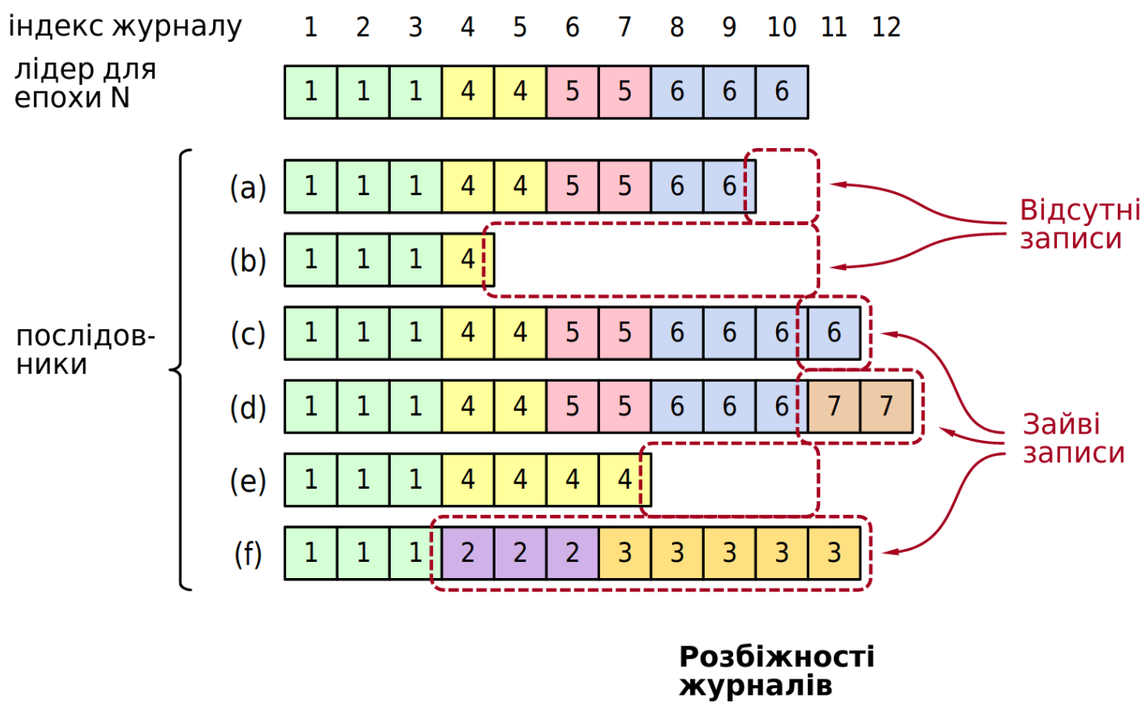
12. J. Kreps, N. Narkhede, J. Rao et al., Kafka: A distributed messaging system for log processing // *NetDB*, 2011.
13. Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. – СПб.: Питер, 2018. – 640 с.: ил. – (Серия «Бестселлеры O'Reilly»).
14. L. Lamport. The part-time parliament // ACM Transactions on Computer Systems (TOCS), Vol. 16, 1998 - pp. 133-169.
15. L. Lamport. Time, Clocks, and Ordering of Events in a Distributed System // Communications of tha ACM, Vol.21, 1978 – pp. 558-565
16. L. Lamport, "Paxos made simple", *ACM SIGACT News*, vol. 32, no. 4, pp. 18-25, 2001.
17. L. Lamport, "Generalized consensus and paxos", *Technical Report MSR-TR-2005-33 Microsoft Research Tech. Rep.*, 2005.
18. LAMPSON B.W. The ABCD's of Paxos // InProc.PODC'01, ACM Symposium on Principles of DistributedComputing, ACM,2001 – pp.13–13.
19. M. Burrows, "The chubby lock service for loosely-coupled distributed systems" in OSDI, USENIX Association, pp. 335-350, 2006.
20. M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process // Journal of the Assccktion for Computing Machinery, Vol. 32, No. 2, April 1985 - pp. 374-382.
21. Preethi Kasireddy. How Does Distributed Consensus Work? - [Електроний ресурс] - Режим доступу: https://medium.com/s/story/lets-take-a-crack-at-understanding-distributed-consensus-dad23d0dc95
22. Paul Krzyzanowski. Understanding Paxos - [Електроний ресурс] - Режим доступу: https://www.cs.rutgers.edu/~pxk/417/notes/paxos.html
23. P. Hunt, M. Konar, F. Junqueira, B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems", *USENIX ATC*, vol. 10, 2010.
24. RabbitMQ Documentation - [Електроний ресурс] - Режим доступу: https://www.rabbitmq.com/queues.html

25. R. Van Renesse, D. Altinbuken, "Paxos made moderately complex", *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 42, 2015.
26. T. Yovtchev - SQS FIFO Queues: Message Ordering and Exactly-Once Processing Guaranteed? - [Електроний ресурс] - Режим доступу: https://www.ably.io/blog/sqs-fifo-queues-message-ordering-and-exactly-once-processing-guaranteed/
27. Y. Saito, M. Shapiro, "Optimistic replication", *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42-81, 2005.

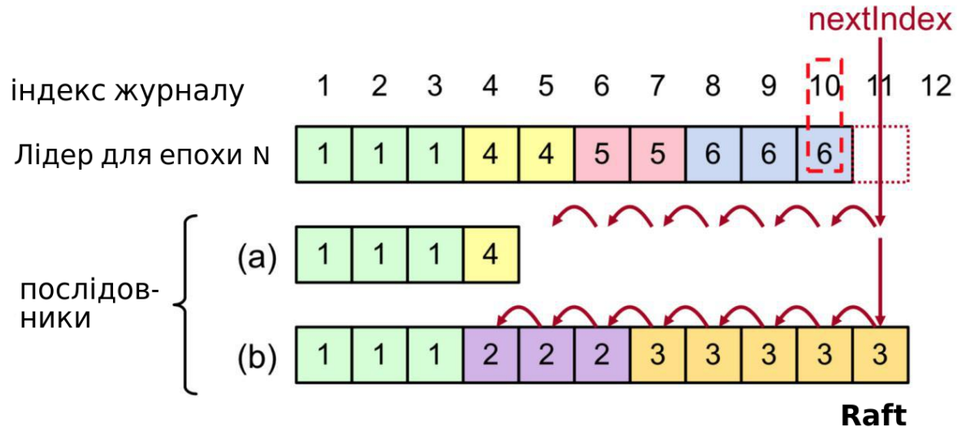
ДОДАТОК А

Графічні матеріали

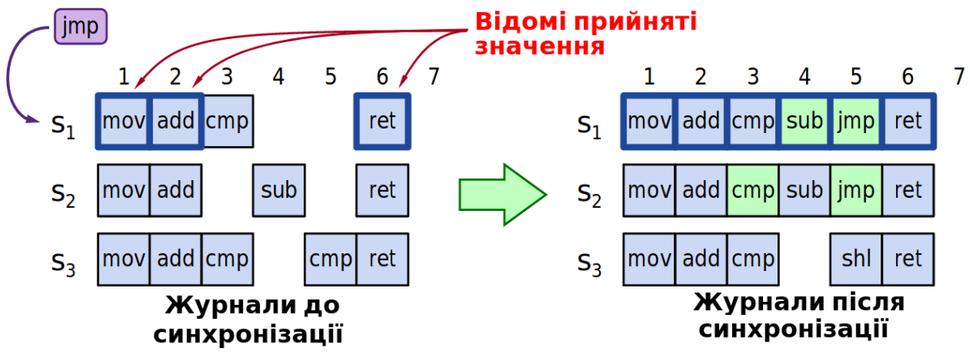
Алгоритм усунення розбіжностей в журналах лідера і послідовників



Розбіжності журналів



Raft



Журнали до синхронізації

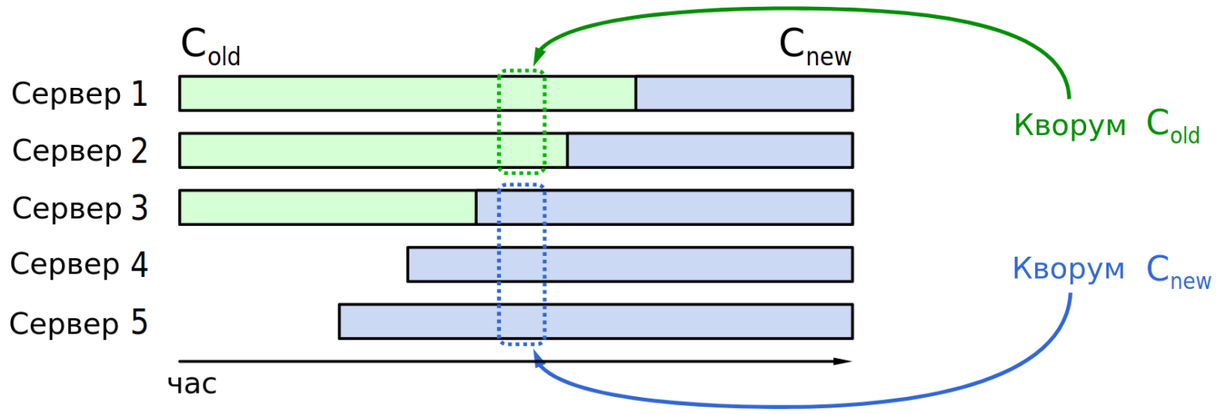
Журнали після синхронізації

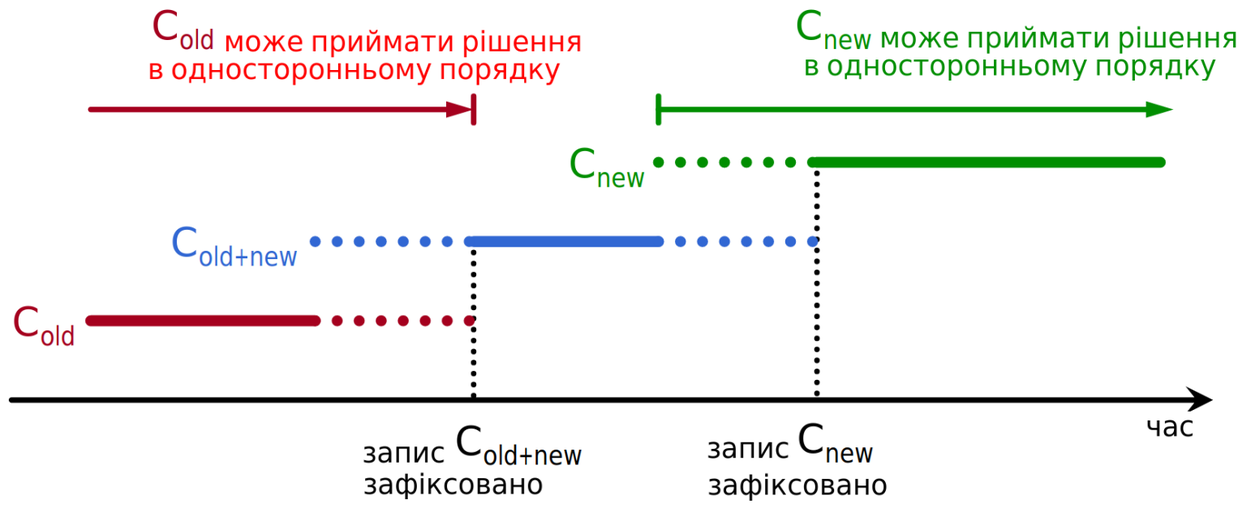
Paxos

Демонстраційний плакат 1
до магістерської дисертації на тему
«Гарантований порядок доставки повідомлень в хмарних системах»

Розробила: Шайдурова К.А.
Прийняв: _____

Алгоритм введення в дію нової конфігурації кластеру

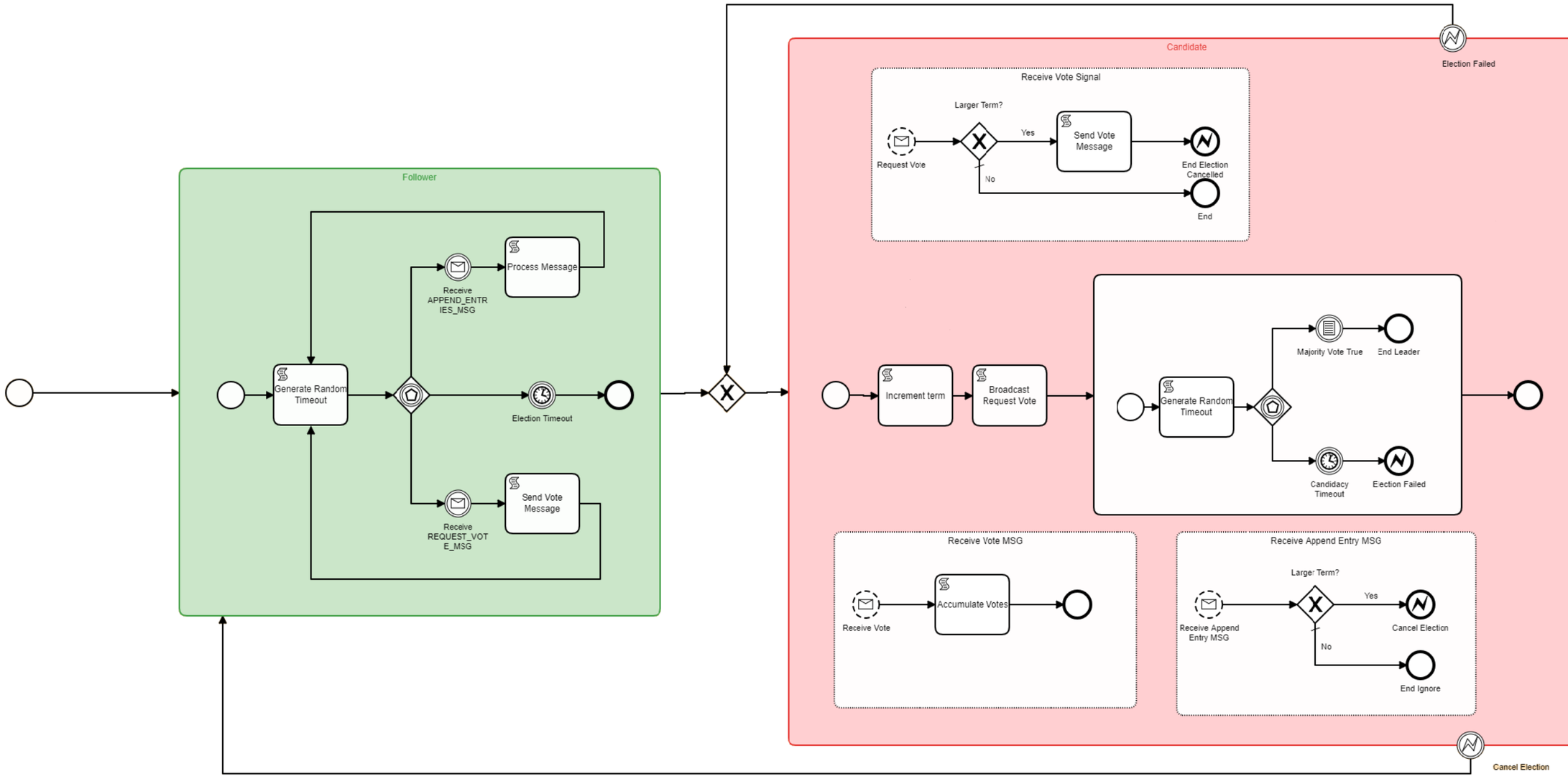




Демонстраційний плакат 2
до магістерської дисертації на тему
«Гарантований порядок доставки повідомлень в хмарних системах»

Розробила: Шайдурова К.А.
Прийняв: _____

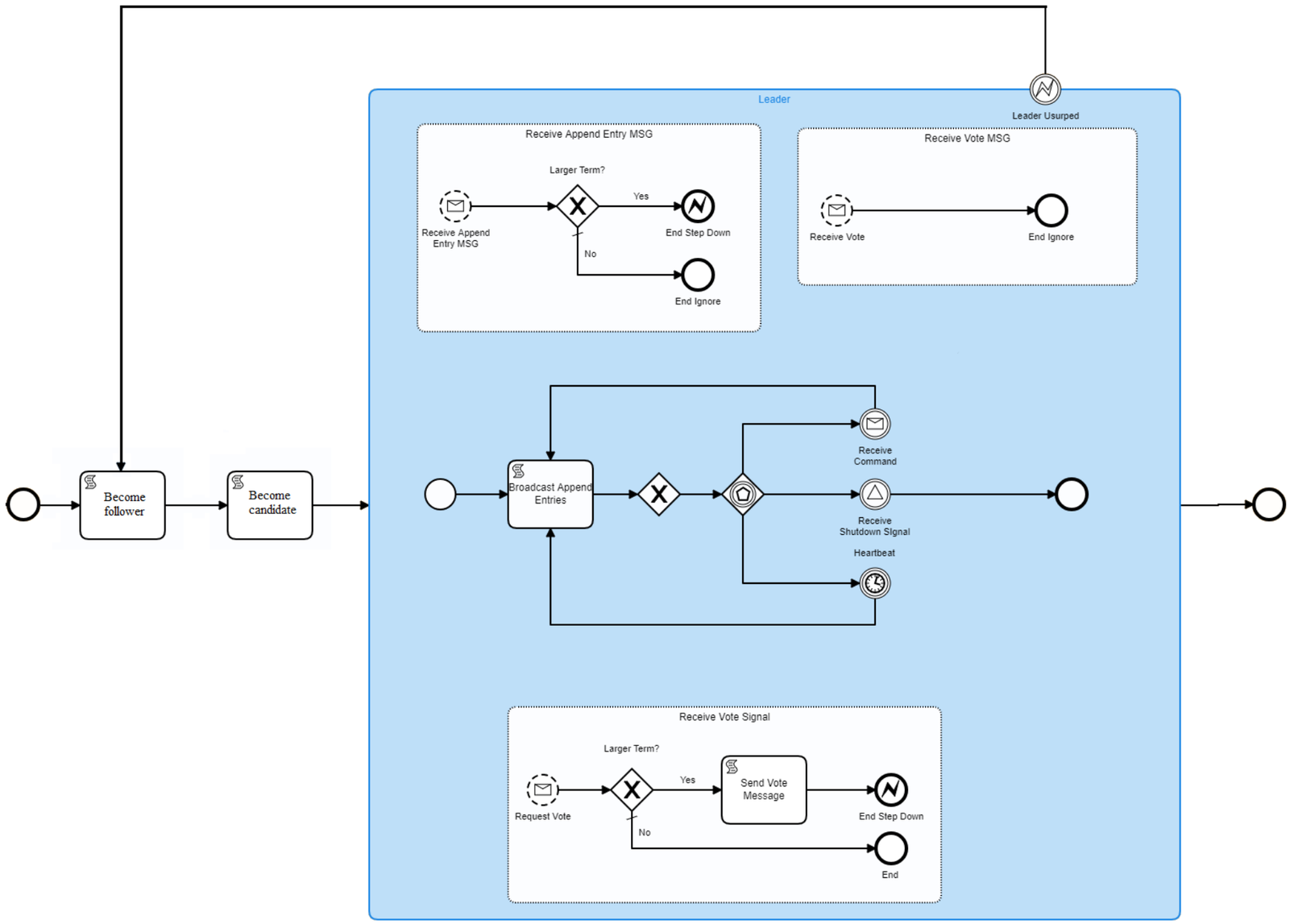
BPMN-діаграма переходу зі стану послідовника до кандидата



Демонстраційний плакат 3
до магістерської дисертації на тему
«Гарантований порядок доставки повідомлень в хмарних системах»

Розробила: Шайдурова К.А.
Прийняв: _____

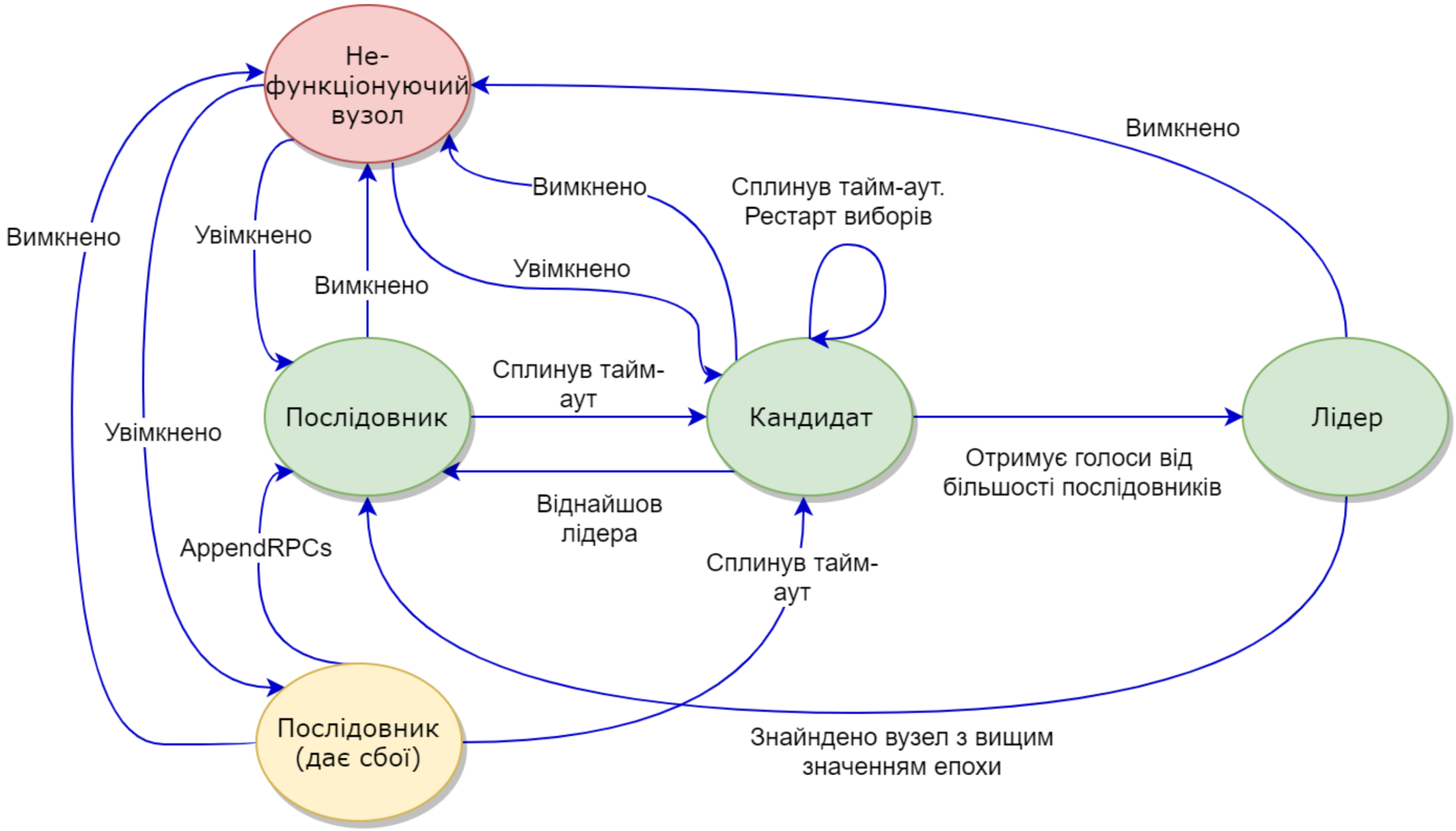
BPMN-діаграма роботи лідера



Демонстраційний плакат 4
до магістерської дисертації на тему
«Гарантований порядок доставки повідомлень в хмарних системах»

Розробила: Шайдурова К.А.
Прийняв: _____

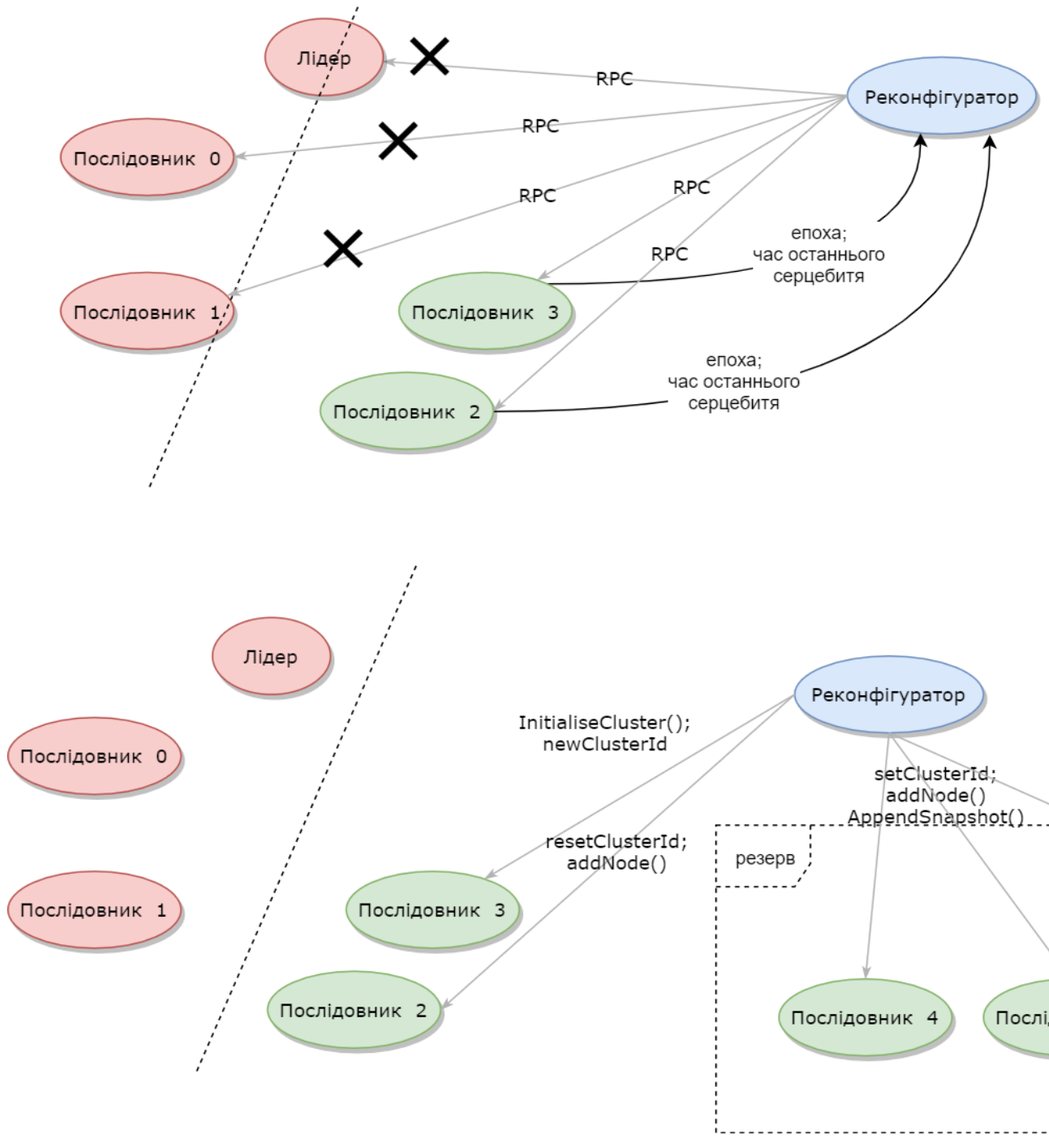
Алгоритм перевиборів Raft



Демонстраційний плакат 5
до магістерської дисертації на тему
«Гарантований порядок доставки повідомлень в хмарних системах»

Розробила: Шайдурова К.А.
Прийняв: _____

Алгоритм переконфігурації кластеру після втрати кворуму та лідера



Демонстраційний плакат 6
до магістерської дисертації на тему
«Гарантований порядок доставки повідомлень в хмарних системах»

Розробила: Шайдурова К.А.
Прийняв: _____

ДОДАТОК Б

Результат перевірки на співпадіння